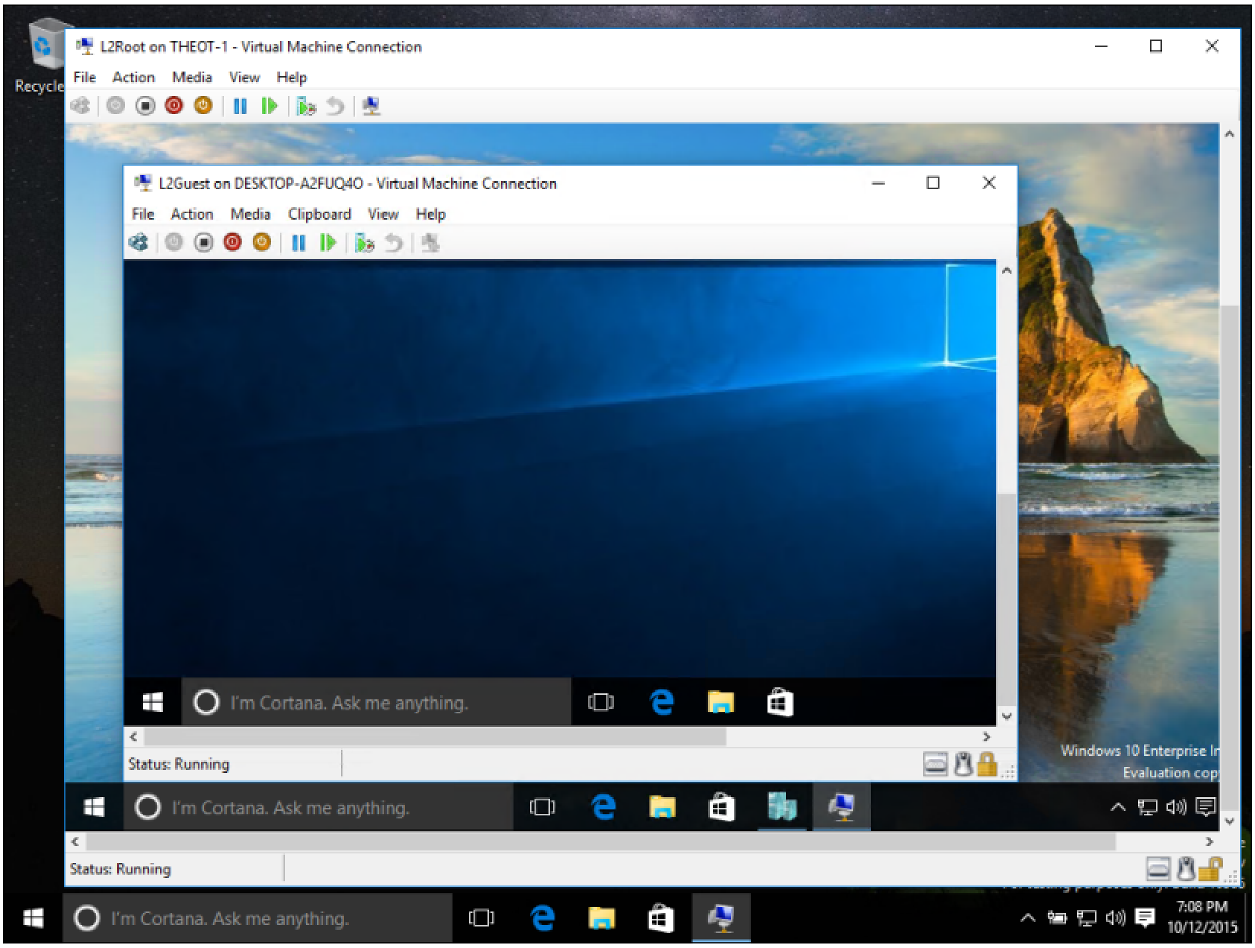
Introduction to Hyper-V on Windows 10

Article • 04/26/2022

Whether you are a software developer, an IT professional, or a technology enthusiast, many of you need to run multiple operating systems. Hyper-V lets you run multiple operating systems as virtual machines on Windows.



Hyper-V specifically provides hardware virtualization. That means each virtual machine runs on virtual hardware. Hyper-V lets you create virtual hard drives, virtual switches, and a number of other virtual devices all of which can be added to virtual machines.

Reasons to use virtualization

Virtualization allows you to:

- Run software that requires an older versions of Windows or non-Windows operating systems.
- Experiment with other operating systems. Hyper-V makes it very easy to create and remove different operating systems.

- Test software on multiple operating systems using multiple virtual machines. With Hyper-V, you can run them all on a single desktop or laptop computer. These virtual machines can be exported and then imported into any other Hyper-V system, including Azure.

System requirements

Hyper-V is available on 64-bit versions of Windows 10 Pro, Enterprise, and Education. It is not available on the Home edition.

> Upgrade from Windows 10 Home edition to Windows 10 Pro by opening **Settings** > **Update and Security** > **Activation**. Here you can visit the store and purchase an upgrade.

Most computers run Hyper-V, however each virtual machine runs a completely separate operating system. You can generally run one or more virtual machines on a computer with 4GB of RAM, though you'll need more resources for additional virtual machines or to install and run resource intense software like games, video editing, or engineering design software.

For more information about Hyper-V's system requirements and how to verify that Hyper-V runs on your machine, see the Hyper-V Requirements Reference.

Operating systems you can run in a virtual machine

Hyper-V on Windows supports many different operating systems in a virtual machine including various releases of Linux, FreeBSD, and Windows.

As a reminder, you'll need to have a valid license for any operating systems you use in the VMs.

For information about which operating systems are supported as guests in Hyper-V on Windows, see Supported Windows Guest Operating Systems and Supported Linux Guest Operating Systems.

Differences between Hyper-V on Windows and Hyper-V on Windows Server

There are some features that work differently in Hyper-V on Windows than they do in Hyper-V running on Windows Server.

Hyper-V features only available on Windows Server:

- Live migration of virtual machines from one host to another
- Hyper-V Replica
- Virtual Fiber Channel
- SR-IOV networking
- Shared .VHDX

Hyper-V features only available on Windows 10:

- Quick Create and the VM Gallery
- Default network (NAT switch)

The memory management model is different for Hyper-V on Windows. On a server, Hyper-V memory is managed with the assumption that only the virtual machines are running on the server. In Hyper-V on Windows, memory is managed with the expectation that most client machines are running software on host in addition to running virtual machines.

Limitations

Programs that depend on specific hardware will not work well in a virtual machine. For example, games or applications that require processing with GPUs might not work well. Also, applications relying on sub-10ms timers such as live music mixing applications or high precision times could have issues running in a virtual machine.

In addition, if you have Hyper-V enabled, those latency-sensitive, high-precision applications may also have issues running in the host. This is because with virtualization enabled, the host OS also runs on top of the Hyper-V virtualization layer, just as guest operating systems do. However, unlike guests, the host OS is special in that it has direct access to all the hardware, which means that applications with special hardware requirements can still run without issues in the host OS.

Next step

[Install Hyper-V on Windows 10]

Install Hyper-V on Windows 10

Article • 04/26/2022

Enable Hyper-V to create virtual machines on Windows 10.
Hyper-V can be enabled in many ways including using the Windows 10 control panel, PowerShell or using the Deployment Imaging Servicing and Management tool (DISM). This documents walks through each option.

> **Note:** Hyper-V is built into Windows as an optional feature -- there is no Hyper-V download.

Check Requirements

- Windows 10 Enterprise, Pro, or Education
- 64-bit Processor with Second Level Address Translation (SLAT).
- CPU support for VM Monitor Mode Extension (VT-c on Intel CPUs).
- Minimum of 4 GB memory.

The Hyper-V role **cannot** be installed on Windows 10 Home.

Upgrade from Windows 10 Home edition to Windows 10 Pro by opening up **Settings** > **Update and Security** > **Activation**.

For more information and troubleshooting, see Windows 10 Hyper-V System Requirements.

Enable Hyper-V using PowerShell

1. Open a PowerShell console as Administrator.

2. Run the following command:

PowerShell

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All
```

If the command couldn't be found, make sure you're running PowerShell as Administrator.

When the installation has completed, reboot.

Enable Hyper-V with CMD and DISM

The Deployment Image Servicing and Management tool (DISM) helps configure Windows and Windows images. Among its many applications, DISM can enable Windows features while the operating system is running.

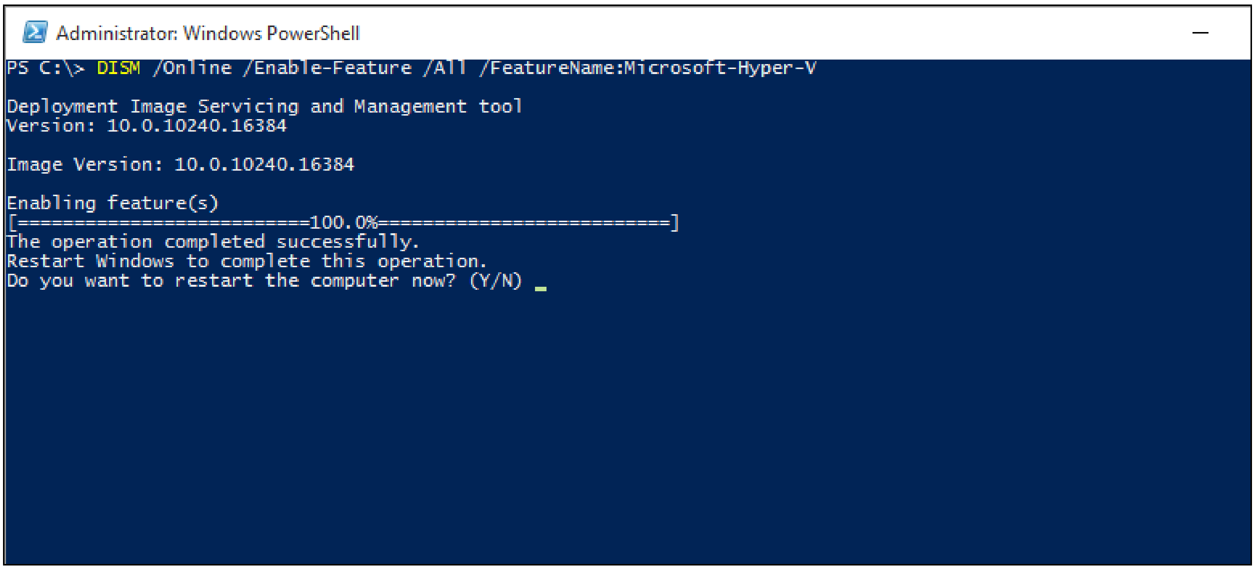
To enable the Hyper-V role using DISM:

1. Open up a PowerShell or CMD session as Administrator.

2. Type the following command:

PowerShell

```
DISM /Online /Enable-Feature /All /FeatureName:Microsoft-Hyper-V
```



For more information about DISM, see the DISM Technical Reference.

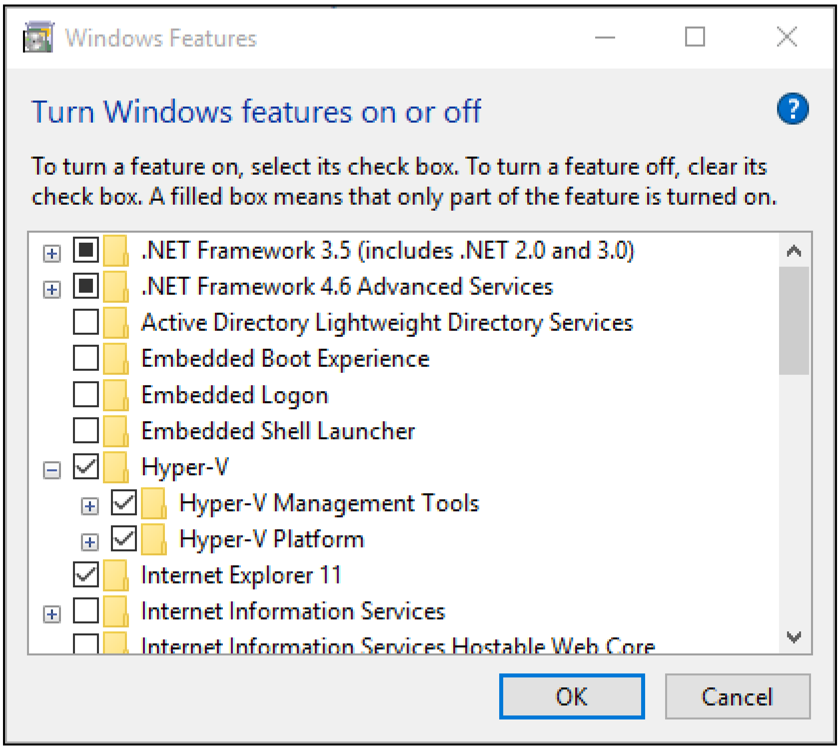
Enable the Hyper-V role through Settings

1. Right click on the Windows button and select 'Apps and Features'.

2. Select **Programs and Features** on the right under related settings.

3. Select **Turn Windows Features on or off**.

4. Select **Hyper-V** and click **OK**.



When the installation has completed you are prompted to restart your computer.

Make virtual machines

[Create your first virtual machine]

Create a Virtual Machine with Hyper-V on Windows 10 Creators Update

Article • 04/26/2022

Create a virtual machine and install its operating system.

We've been building new tools for creating virtual machines so the instructions have changed significantly over the past three releases.

Pick your operating system for the right set of instructions:

- Windows 10 Fall Creators Update (v1709) and later
- Windows 10 Creators Update (v1703)
- Windows 10 Anniversary Update (v1607) and earlier

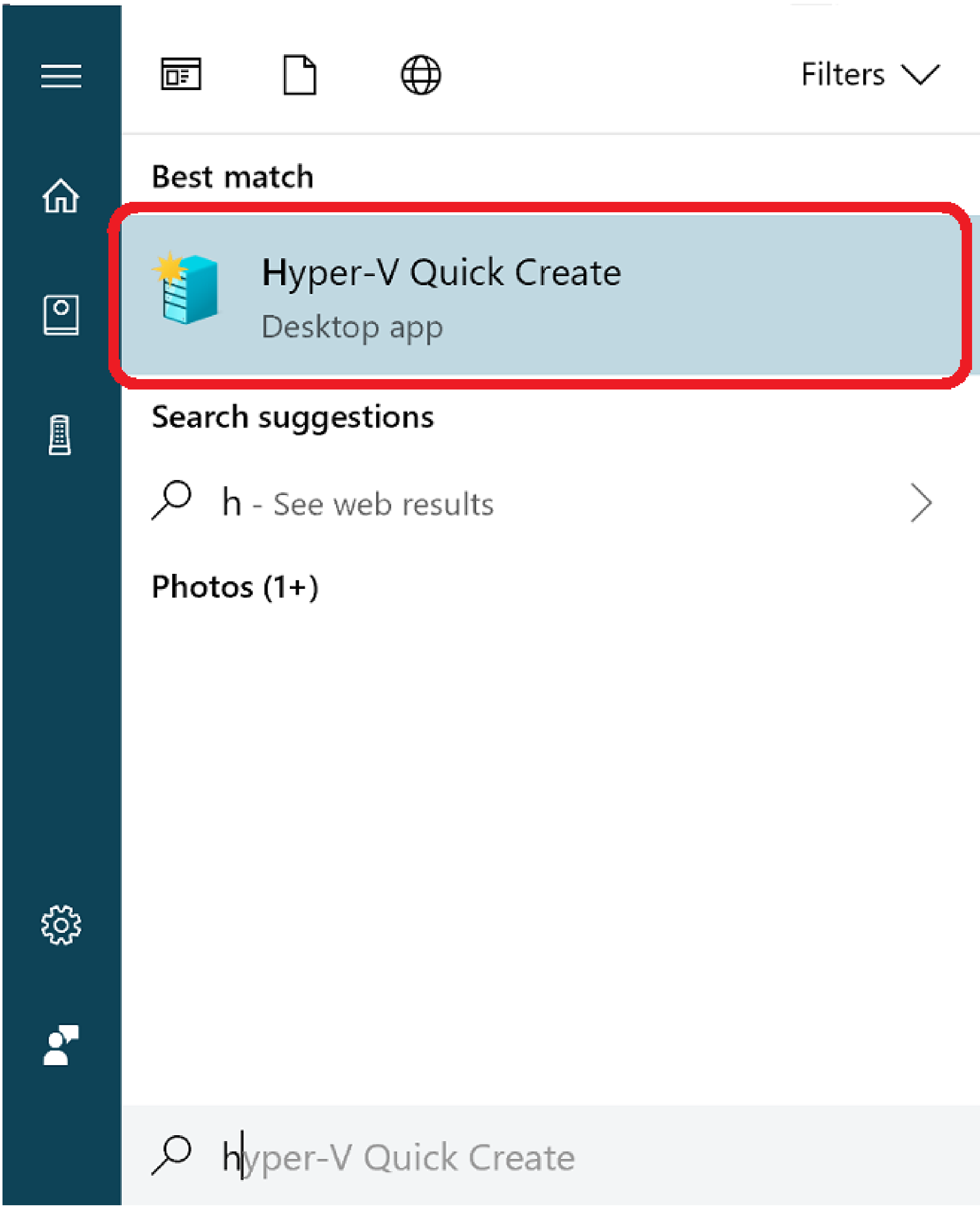
Let's get started.

Windows 10 Fall Creators Update (Windows 10 version 1709)

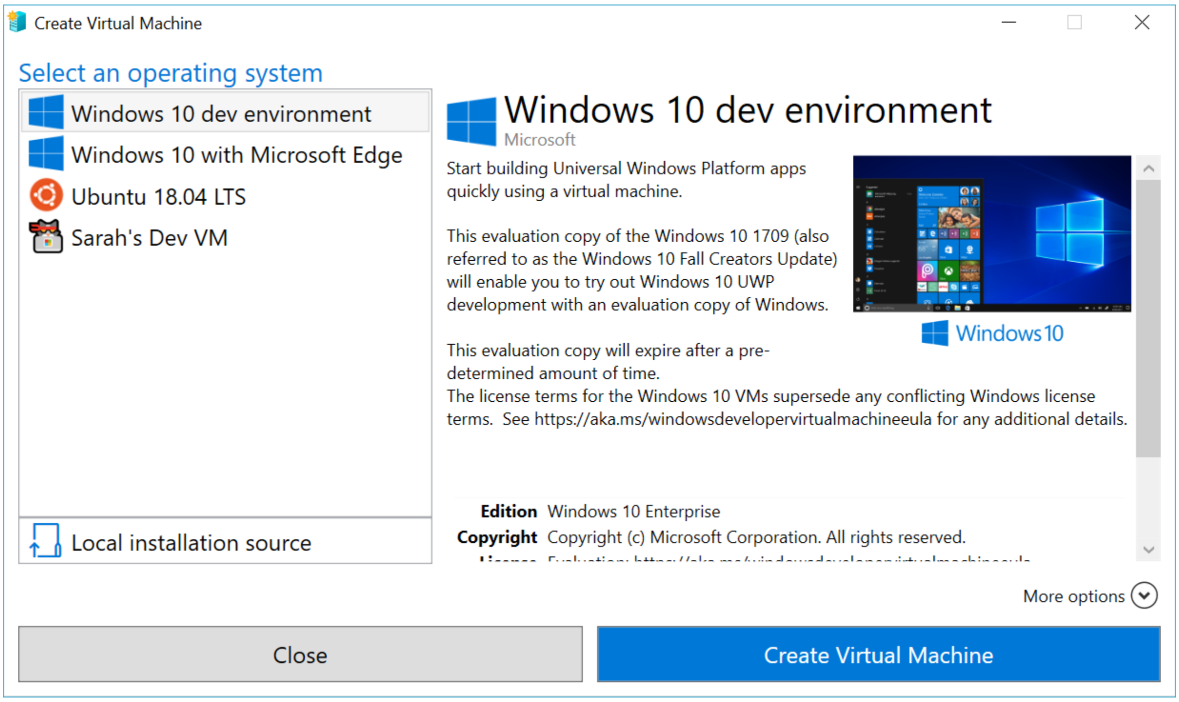
In Fall Creators Update, Quick Create expanded to include a virtual machine gallery that can be launched independently from Hyper-V Manager.

To create a new virtual machine in Fall Creators Update:

1. Open Hyper-V Quick Create from the start menu.

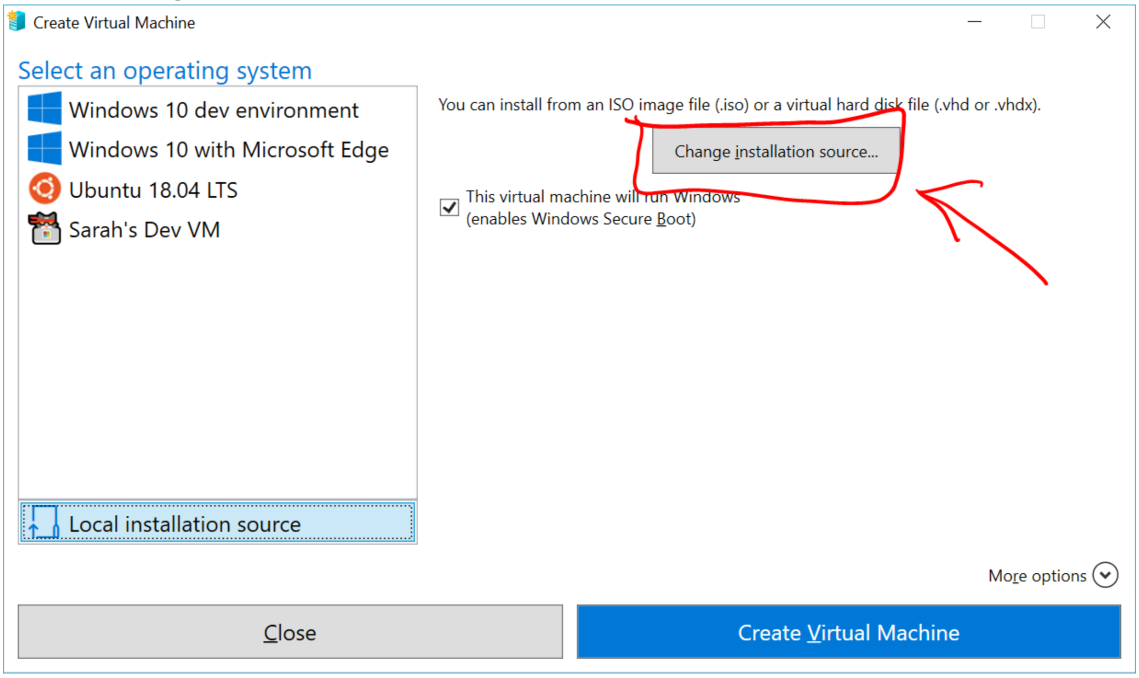


2. Select an operating system or choose your own by using a local installation source.



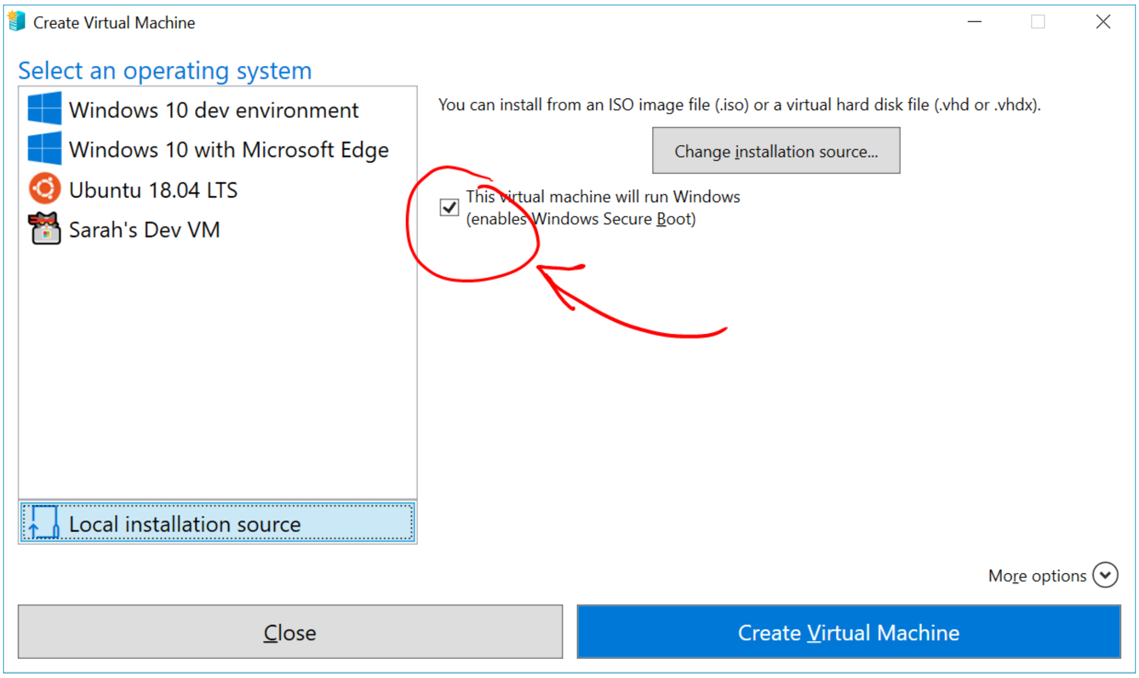
a. If you want to use your own image to create the virtual machine, select **Local Installation Source**.

b. Select **Change Installation Source**.



c. Pick the .iso or .vhdx that you want to turn into a new virtual machine.

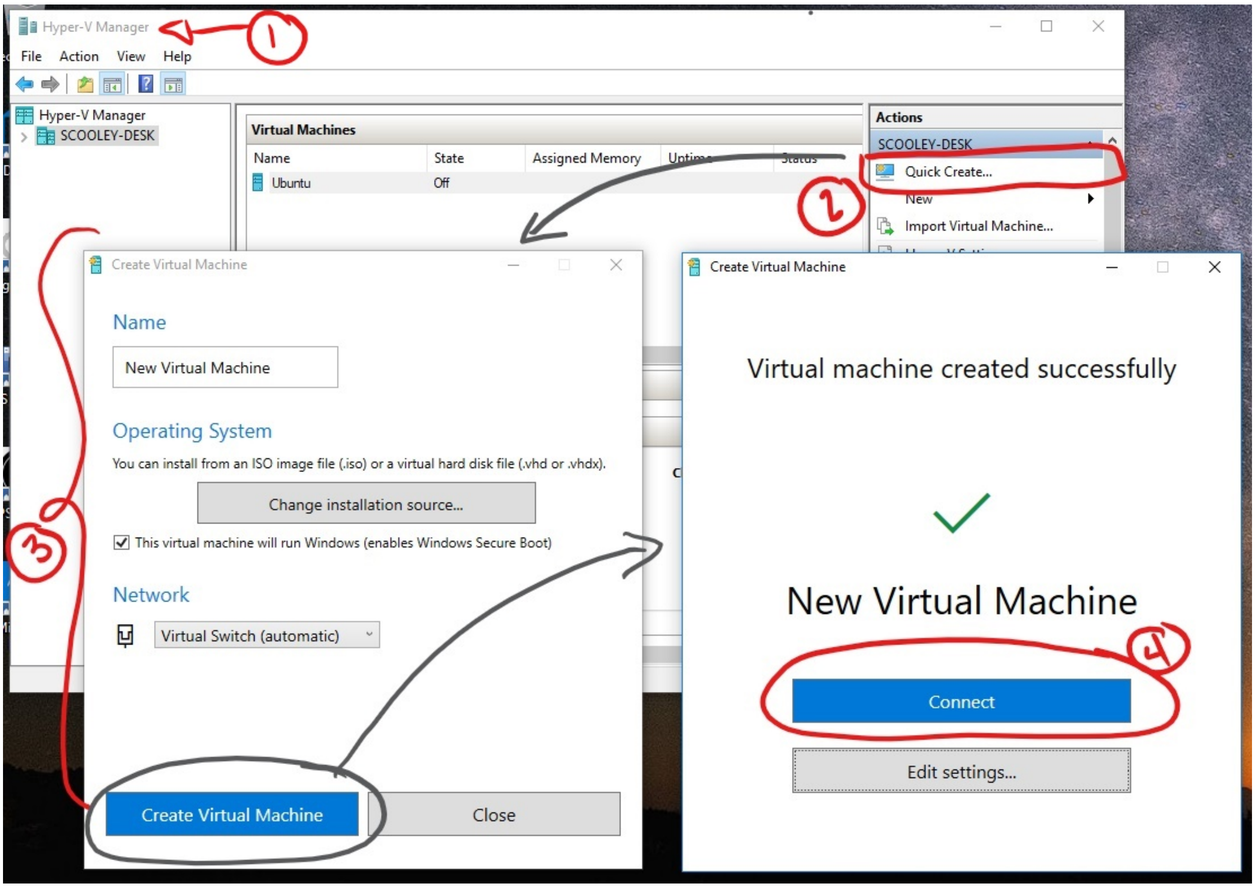
d. If the image is a Linux image, deselect the Secure Boot option.



3. Select "Create Virtual Machine"

That's it! Quick Create will take care of the rest.

Windows 10 Creators Update (Windows 10 version 1703)



1. Open Hyper-V Manager from the start menu.

2. In Hyper-V Manager, Find **Quick Create** in the right hand **Actions** menu.

3. Customize your virtual machine.

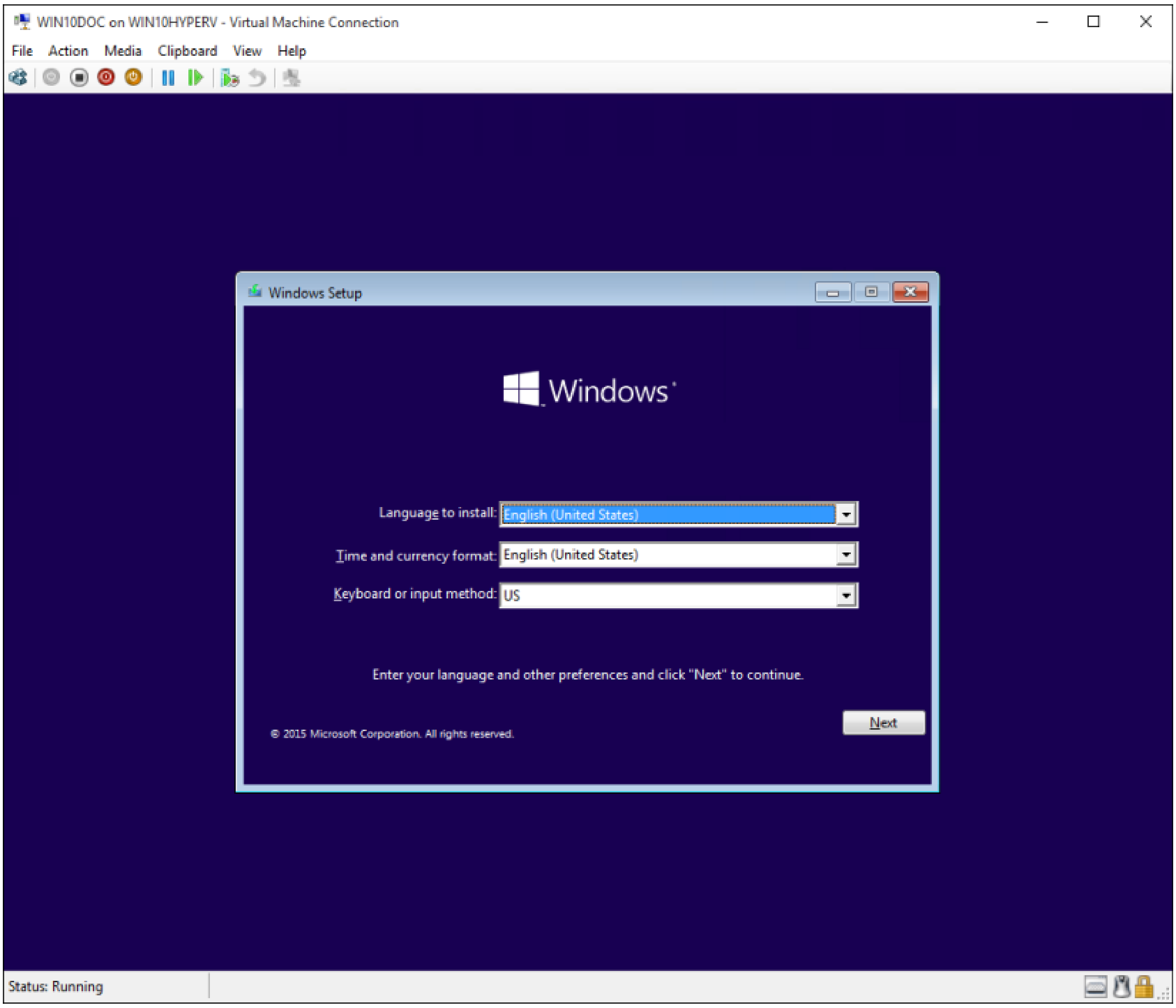
   - (optional) Give the virtual machine a name.
   - Select the installation media for the virtual machine. You can install from a .iso or .vhdx file. If you are installing Windows in the virtual machine, you can enable Windows Secure Boot. Otherwise leave it unselected.
   - Set up network. If you have an existing virtual switch, you can select in the network dropdown. If you have no existing switch, you will see a button to set up an automatic network, which will automatically configure a virtual network.

4. Click **Connect** to start your virtual machine. Don't worry about editing the settings, you can go back and change them any time.

   You may be prompted to ‘Press any key to boot from CD or DVD’. Go ahead and do so. As far as it knows, you're installing from a CD.

Congratulations, you have a new virtual machine. Now you're ready to install the operating system.

Your virtual machine should look something like this:



> **Note:** Unless you're running a volume-licensed version of Windows, you need a separate license for Windows running inside a virtual machine. The virtual machine's operating system is independent of the host operating system.

Before Windows 10 Creators Update (Windows 10 version 1607 and earlier)

If you aren't running Windows 10 Creators Update or later, follow these instructions using New Virtual Machine Wizard instead:

1. Create a virtual network
2. Create a new virtual machine

Working with Hyper-V and Windows PowerShell

Article • 04/26/2022

Now that you have walked through the basics of deploying Hyper-V, creating virtual machines and managing these virtual machines, let's explore how you can automate many of these activities with PowerShell.

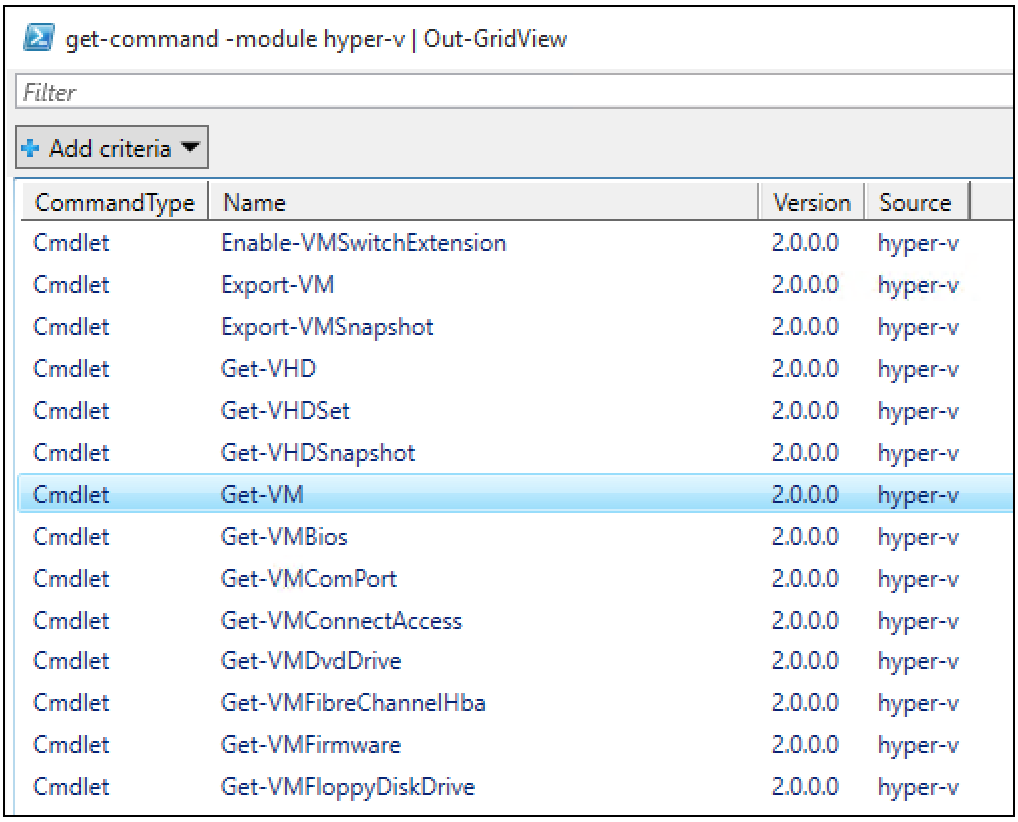
Return a list of Hyper-V commands

1. Click on the Windows start button, type **PowerShell**.
2. Run the following command to display a searchable list of PowerShell commands available with the Hyper-V PowerShell Module.

PowerShell

```
Get-Command -Module hyper-v | Out-GridView
```

You get something like this:



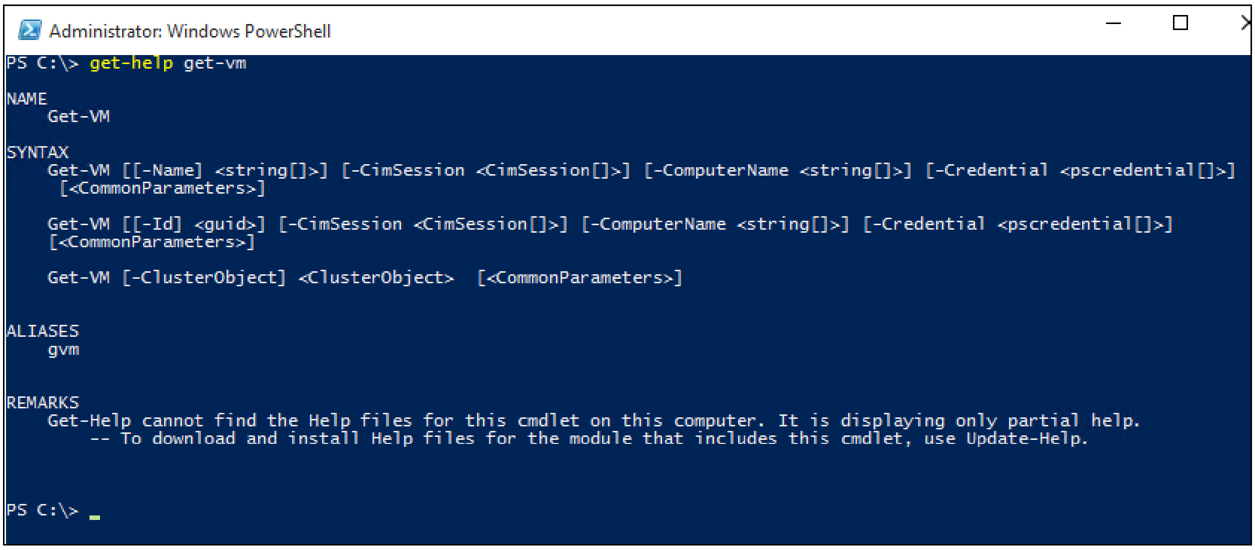
CommandType	Name	Version	Source
Cmdlet	Enable-VMSwitchExtension	2.0.0.0	hyper-v
Cmdlet	Export-VM	2.0.0.0	hyper-v
Cmdlet	Export-VMSnapshot	2.0.0.0	hyper-v
Cmdlet	Get-VHD	2.0.0.0	hyper-v
Cmdlet	Get-VHDSet	2.0.0.0	hyper-v
Cmdlet	Get-VHDSnapshot	2.0.0.0	hyper-v
Cmdlet	Get-VM	2.0.0.0	hyper-v
Cmdlet	Get-VMBios	2.0.0.0	hyper-v
Cmdlet	Get-VMComPort	2.0.0.0	hyper-v
Cmdlet	Get-VMConnectAccess	2.0.0.0	hyper-v
Cmdlet	Get-VMDvdDrive	2.0.0.0	hyper-v
Cmdlet	Get-VMFibreChannelHba	2.0.0.0	hyper-v
Cmdlet	Get-VMFirmware	2.0.0.0	hyper-v
Cmdlet	Get-VMFloppyDiskDrive	2.0.0.0	hyper-v

3. To learn more about a particular PowerShell command use `Get-Help`. For instance running the following command returns information about the `Get-VM` Hyper-V command.

PowerShell

```
Get-Help Get-VM
```

The output shows you how to structure the command, what the required and optional parameters are, and the aliases that you can use.



Return a list of virtual machines

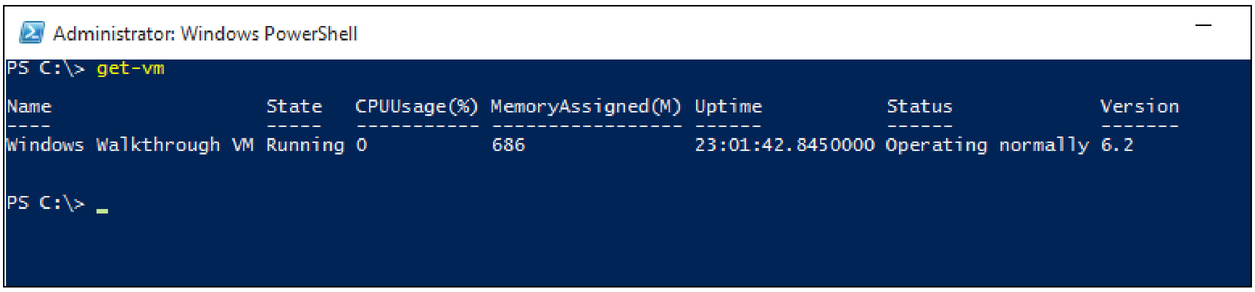
Use the `Get-VM` command to return a list of virtual machines.

1. In PowerShell, run the following command:

PowerShell

```
Get-VM
```

This displays something like this:



2. To return a list of only powered on virtual machines add a filter to the `Get-VM` command. A filter can be added by using the `Where-Object` command. For more information on filtering see the Using the Where-Object documentation.

PowerShell

```
Get-VM | where {$_.State -eq 'Running'}
```

3. To list all virtual machines in a powered off state, run the following command. This command is a copy of the command from step 2 with the filter changed from 'Running' to 'Off'.

PowerShell

```
Get-VM | where {$_.State -eq 'Off'}
```

Start and shut down virtual machines

1. To start a particular virtual machine, run the following command with name of the virtual machine:

PowerShell

```
Start-VM -Name <virtual machine name>
```

2. To start all currently powered off virtual machines, get a list of those machines and pipe the list to the `Start-VM` command:

PowerShell

```
Get-VM | where {$_.State -eq 'Off'} | Start-VM
```

3. To shut down all running virtual machines, run this:

PowerShell

```
Get-VM | where {$_.State -eq 'Running'} | Stop-VM
```

Create a VM checkpoint

To create a checkpoint using PowerShell, select the virtual machine using the `Get-VM` command and pipe this to the `Checkpoint-VM` command. Finally give the checkpoint a name using `-SnapshotName`. The complete command looks like the following:

PowerShell

```
Get-VM -Name <VM Name> | Checkpoint-VM -SnapshotName <name for snapshot>
```

Create a new virtual machine

The following example shows how to create a new virtual machine in the PowerShell Integrated Scripting Environment (ISE). This is a simple example and could be expanded on to include additional PowerShell features and more advanced VM deployments.

1. To open the PowerShell ISE click on start, type **PowerShell ISE**.
2. Run the following code to create a virtual machine. See the New-VM documentation for detailed information on the `New-VM` command.

PowerShell

```
$VMName = "VMNAME"

$VM = @{
    Name = $VMName
    MemoryStartupBytes = 2147483648
    Generation = 2
    NewVHDPath = "C:\Virtual Machines\$VMName\$VMName.vhdx"
    NewVHDSizeBytes = 53687091200
    BootDevice = "VHD"
    Path = "C:\Virtual Machines\$VMName"
    SwitchName = (Get-VMSwitch).Name
}

New-VM @VM
```

Wrap up and References

This document has shown some simple steps to explorer the Hyper-V PowerShell module as well as some sample scenarios. For more information on the Hyper-V PowerShell module, see the Hyper-V Cmdlets in Windows PowerShell reference.

Share devices with your virtual machine

Article • 05/09/2023

> Only available for Windows virtual machines.

Enhanced Session Mode lets Hyper-V connect to virtual machines using RDP (remote desktop protocol). Not only does this improve your general virtual machine viewing experience, connecting with RDP also allows the virtual machine to share devices with your computer. Since it's on by default in Windows 10, you're probably already using RDP to connect to your Windows virtual machines. This article highlights some of the benefits and hidden options in the connection settings dialogue.

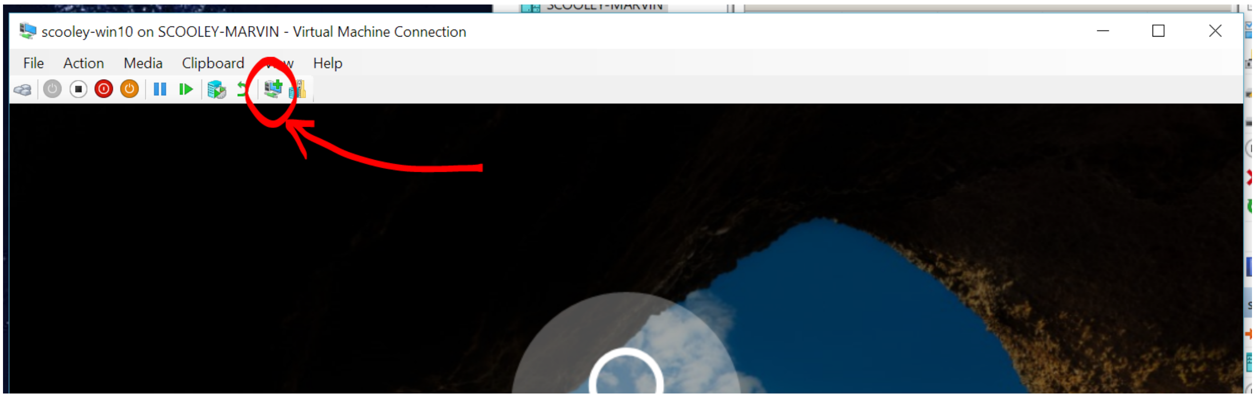
RDP/Enhanced Session mode:

- Makes virtual machines resizable and high DPI aware.
- Improves virtual machine integration
  - Shared clipboard
  - File sharing via drag drop and copy paste
- Allows device sharing
  - Microphone/Speakers
  - USB devices
  - Data disks (including C:)
  - Printers

This article shows you how to see your session type, enter enhanced session mode, and configure your session settings.

Check session type

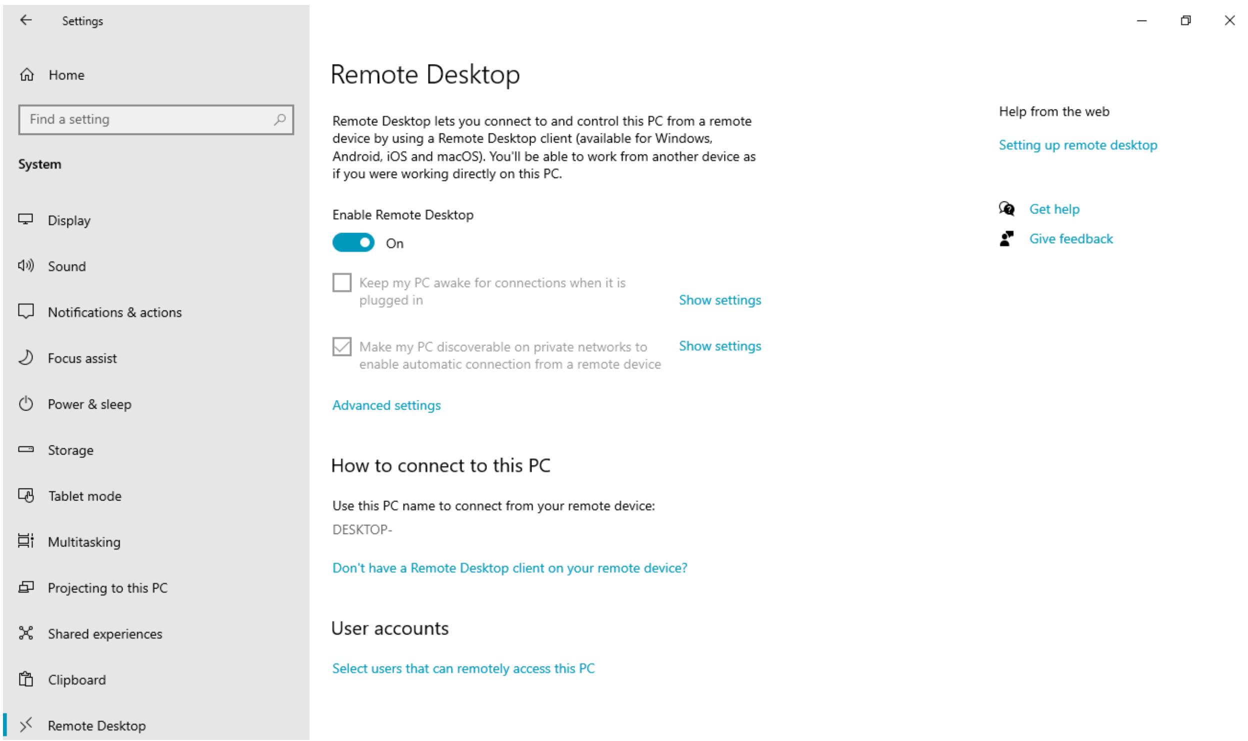
You can check to see what type of connection you have using the Enhanced Session mode icon in the top of the Virtual Machine Connect tool (VMConnect). This button also lets you toggle between basic session and enhanced session mode.



icon	connection state
	You are currently running in enhanced session mode. Clicking this icon will reconnect to your virtual machine in basic mode.
	You are currently running in basic session mode but enhanced session mode is available. Clicking this icon will reconnect to your virtual machine in enhanced session mode.
	You are currently running in basic mode. Enhanced session mode isn't available for this virtual machine.

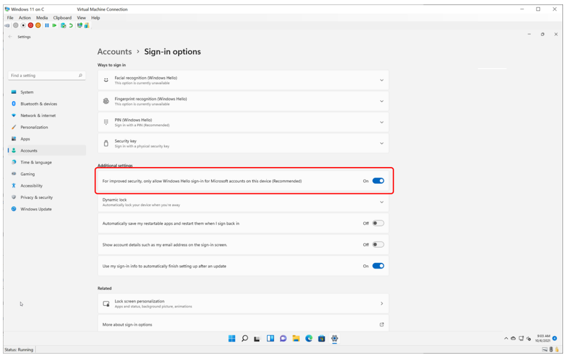
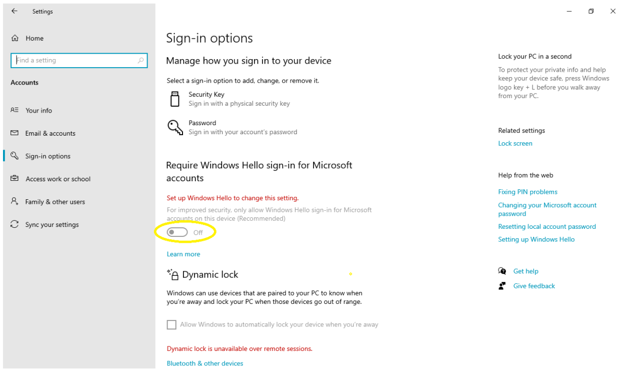
Configure VM for Remote Desktop

Enhanced Session Mode requires Remote Desktop to be enabled in the VM. Search for "Remote Desktop settings" in the Settings app or Start menu. Turn "Enable Remote Desktop" on.



Versions newer than Windows 10, version 2004 will requrire an additional setting, this includes Windows 11. If the Virtual Machine Connect window shows a background without a login prompt, then you need to make one more change.

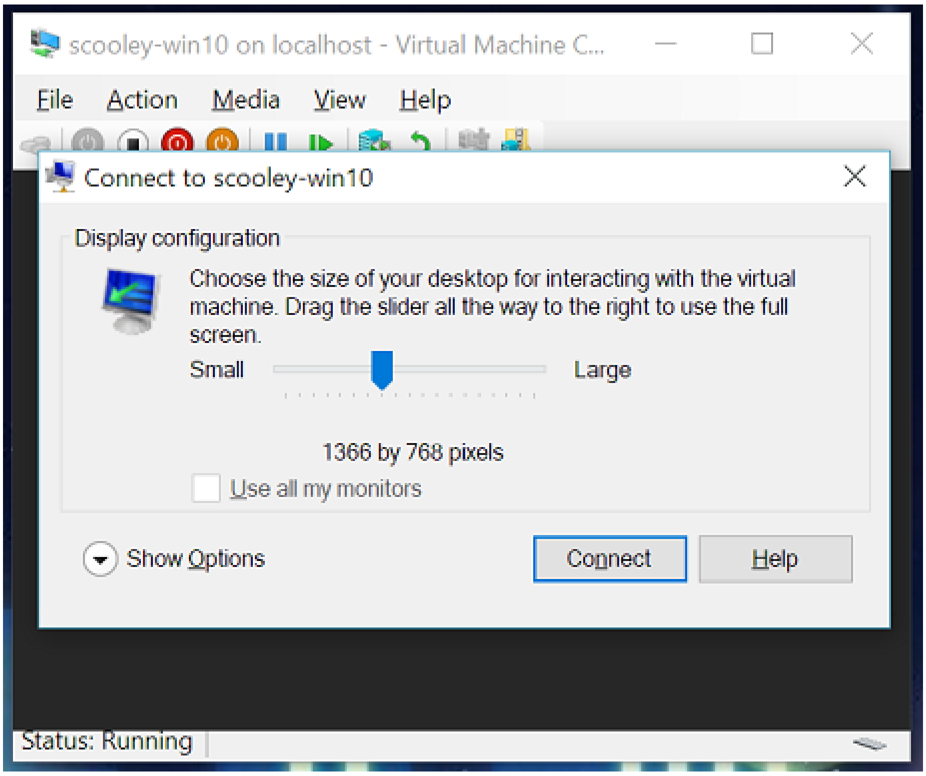
First, log back in to the VM using Basic Mode. Search for "Sign-In Options" in the Settings app or Start menu. On this page, turn "Require Windows Hello sign-in for Microsoft accounts" off.

Windows 11	Windows 10
	

Now, sign out of the VM or reboot before closing the Virtual Machine Connect window.

Share drives and devices

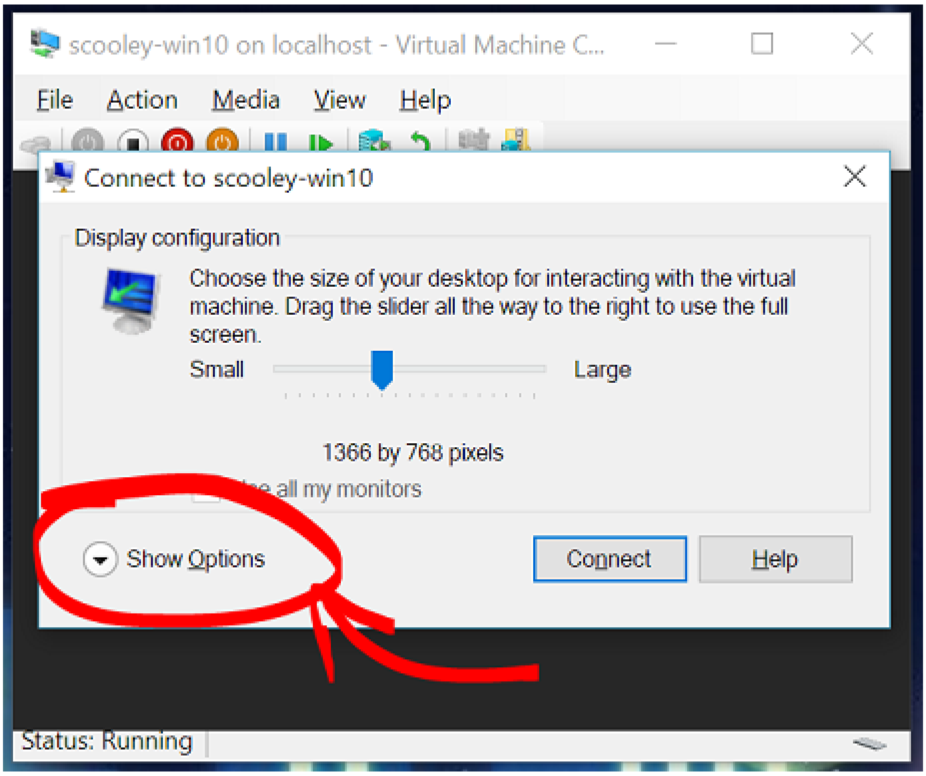
Enhanced Session Mode's device sharing capabilities are hidden inside this inconspicuous connection window that pops up when you connect to a virtual machine:



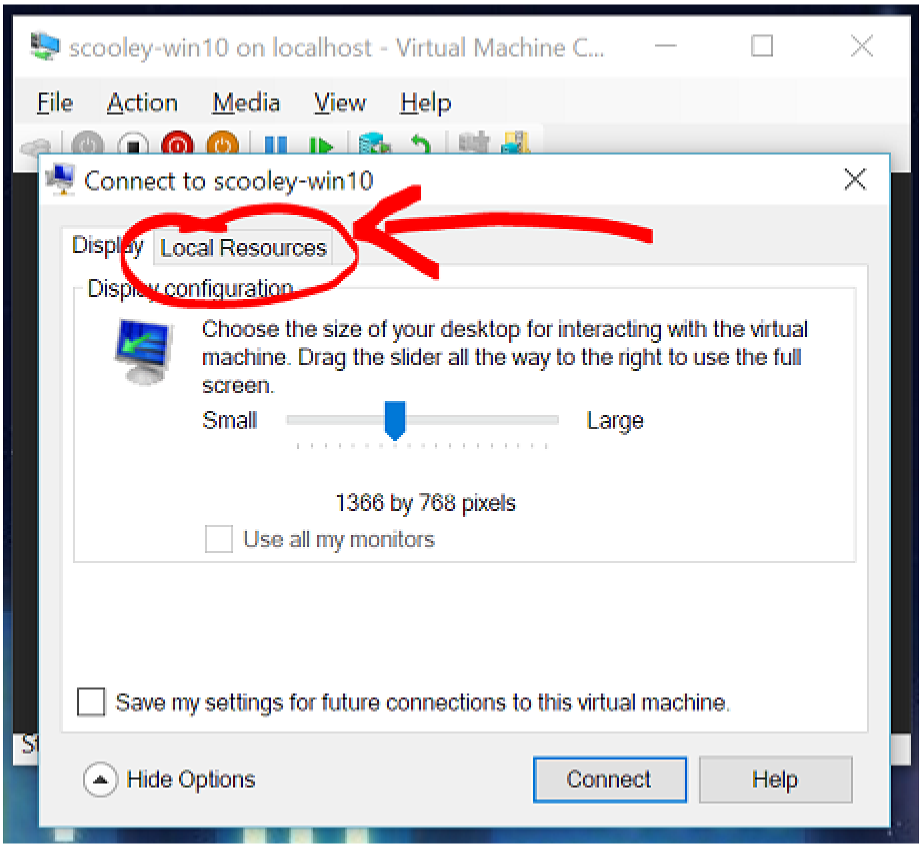
By default, virtual machines using enhanced session mode will share clipboard and printers. They are also configured by default to pass audio from the virtual machine back to your computer's speakers.

To share devices with your virtual machine or to change those default settings:

1. Show more options



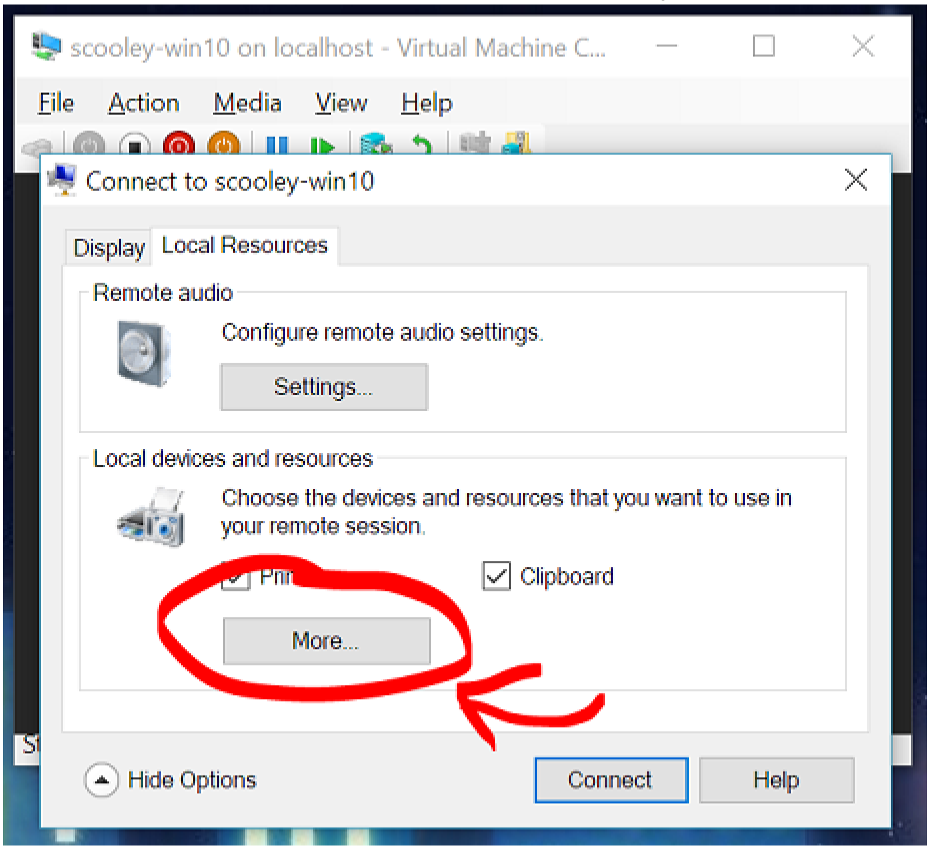
1. View local resources



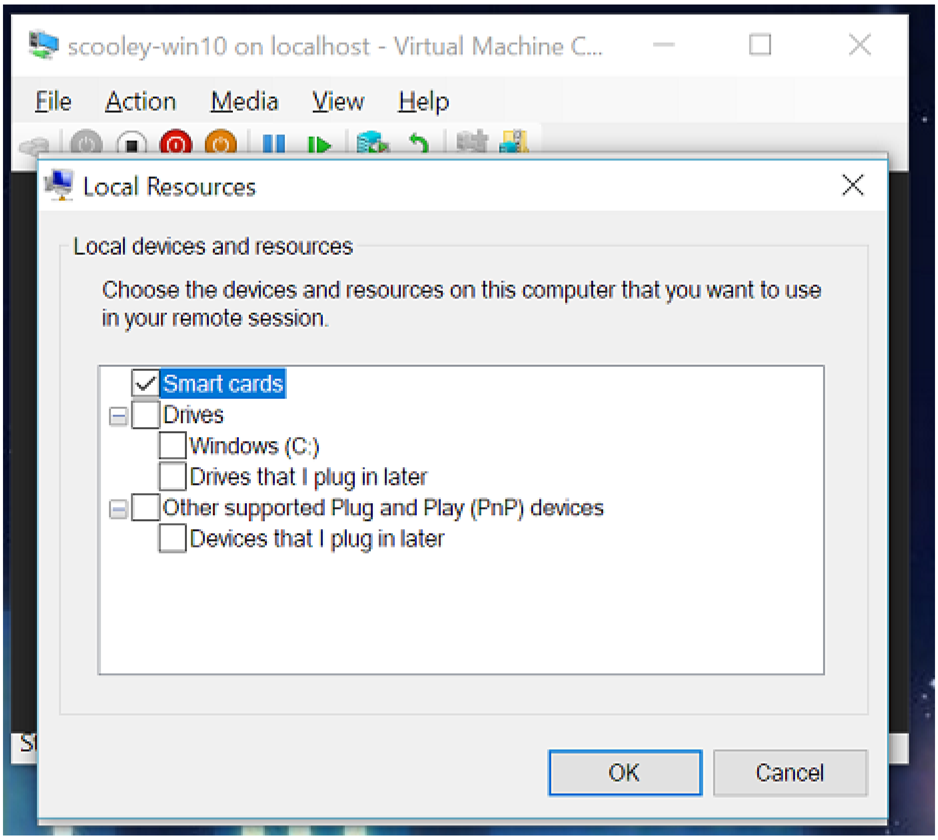
Share storage and USB devices

By default, virtual machines using enhanced session mode share printers, the clipboard, pass smartcard and other security devices through to the virtual machine so you can use more secure login tools from your virtual machine.

To share other devices, such as USB devices or your C: drive, select the "More..." menu:



From there you can select the devices you'd like to share with the virtual machine. The system drive (Windows C:) is especially helpful for file sharing.

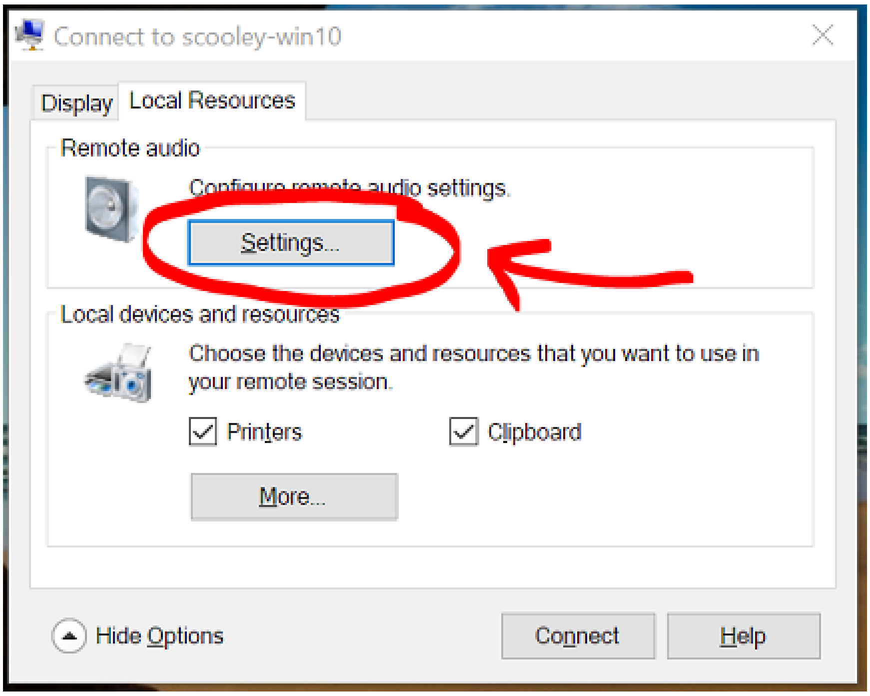


Share audio devices (speakers and microphones)

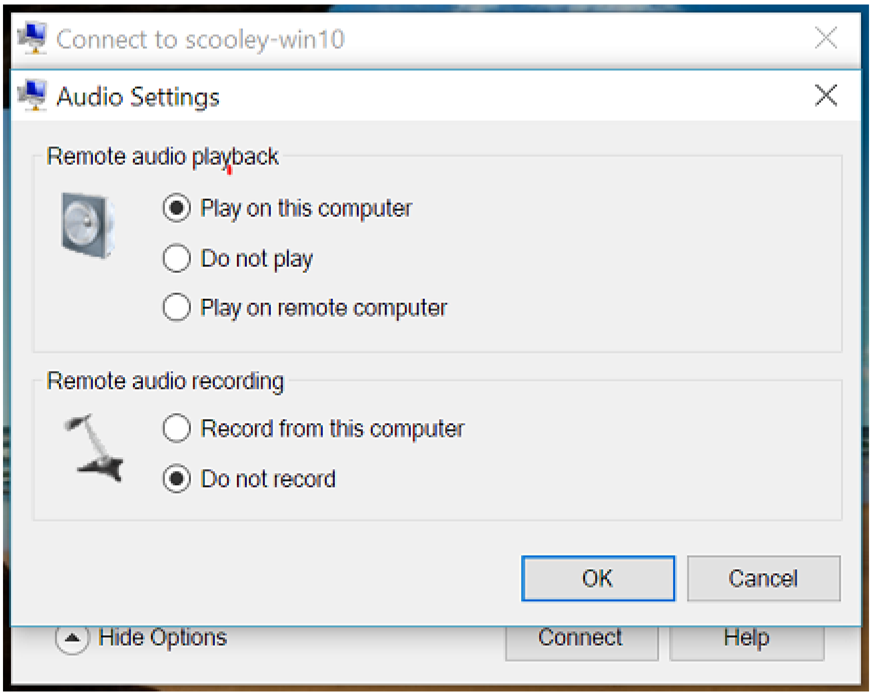
By default, virtual machines using enhanced session mode pass audio through so you can hear audio from the virtual machine. The virtual machine will use the audio device currently selected on the host machine.

To change those settings or to add microphone passthrough (so you can record audio in a virtual machine):

Select the "Settings..." menu for configuring remote audio settings



Now configure audio and microphone settings



Since your virtual machine is probably running locally, the "play on this computer" and "play on remote computer" options will yield the same results.

Re-launching the connection settings

If you aren't getting the resolution and device sharing dialogue box, try launching VMConnect independently from either the Windows menu or from the command line as Administrator.

```Powershell
vmconnect.exe
```

Using checkpoints to revert virtual machines to a previous state

Article • 04/26/2022

One of the great benefits to virtualization is the ability to easily save the state of a virtual machine. In Hyper-V this is done through the use of virtual machine checkpoints. You may want to create a virtual machine checkpoint before making software configuration changes, applying a software update, or installing new software. If a system change were to cause an issue, the virtual machine can be reverted to the state at which it was when then checkpoint was taken.

Windows 10 Hyper-V includes two types of checkpoints:

- **Standard Checkpoints**: takes a snapshot of the virtual machine and virtual machine memory state at the time the checkpoint is initiated. A snapshot is not a full backup and can cause data consistency issues with systems that replicate data between different nodes such as Active Directory. Hyper-V only offered standard checkpoints (formerly called snapshots) prior to Windows 10.

- **Production Checkpoints**: uses Volume Shadow Copy Service or File System Freeze on a Linux virtual machine to create a data-consistent backup of the virtual machine. No snapshot of the virtual machine memory state is taken.

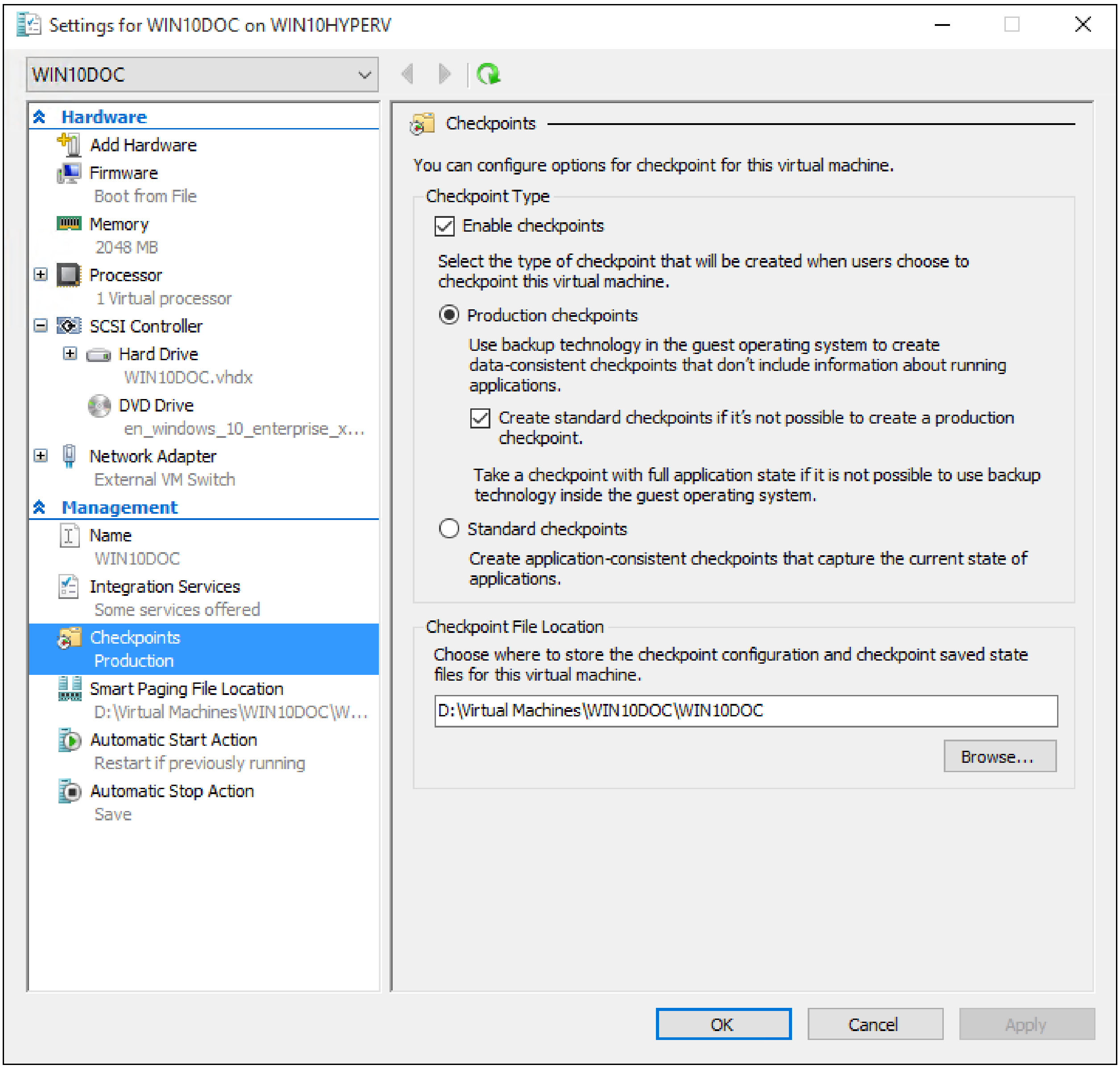
Production checkpoints are selected by default however this can be changed using either Hyper-V manager or PowerShell.

> **Note:** The Hyper-V PowerShell module has several aliases so that checkpoint and snapshot can be used interchangeably.
> This document uses checkpoint, however be aware that you may see similar commands using the term snapshot.

Changing the Checkpoint Type

Using Hyper-V Manager

1. Open Hyper-V Manager.
2. Right click on a virtual machine and select **settings**.
3. Under Management select **Checkpoints**.
4. Select the desired checkpoint type.



Using PowerShell

The following commands can be run to change the checkpoint with PowerShell.

Set to Standard Checkpoint:

PowerShell

```
Set-VM -Name <vmname> -CheckpointType Standard
```

Set to Production Checkpoint, if the production checkpoint fails a standard checkpoint is being created:

PowerShell

```
Set-VM -Name <vmname> -CheckpointType Production
```

Set to Production Checkpoint, if the production checkpoint fails a standard checkpoint is not being created.

```PowerShell
Set-VM -Name <vmname> -CheckpointType ProductionOnly
```

Creating checkpoints

Creates a checkpoint of the type configured for the virtual machine. See the Configuring Checkpoint Type section earlier in this document for instructions on how to change this type.

Using Hyper-V Manager

To create a checkpoint:

1. In Hyper-V Manager, select the virtual machine.
2. Right-click the name of the virtual machine, and then click **Checkpoint**.
3. When the process is complete, the checkpoint will appear under **Checkpoints** in the **Hyper-V Manager**.

Using PowerShell

Create a checkpoint using the **CheckPoint-VM** command.

```PowerShell
Checkpoint-VM -Name <VMName>
```

When the checkpoint process has completed, view a list of checkpoints for a virtual machine use the **Get-VMCheckpoint** command.

```PowerShell
Get-VMCheckpoint -VMName <VMName>
```

Applying checkpoints

If you want to revert your virtual machine to a previous point-in-time, you can apply an existing checkpoint.

Using Hyper-V Manager

1. In **Hyper-V Manager**, under **Virtual Machines**, select the virtual machine.
2. In the Checkpoints section, right-click the checkpoint that you want to use and click **Apply**.
3. A dialog box appears with the following options:

- **Create Checkpoint and Apply**: Creates a new checkpoint of the virtual machine before it applies the earlier checkpoint.
- **Apply**: Applies only the checkpoint that you have chosen. You cannot undo this action.
- **Cancel**: Closes the dialog box without doing anything.

Select either Apply option to create apply the checkpoint.

Using PowerShell

5. To see a list of checkpoints for a virtual machine use the **Get-VMCheckpoint** command.

PowerShell

```
Get-VMCheckpoint -VMName <VMName>
```

6. To apply the checkpoint use the **Restore-VMCheckpoint** command.

PowerShell

```
Restore-VMCheckpoint -Name <checkpoint name> -VMName <VMName> -
Confirm:$false
```

Renaming checkpoints

Many checkpoints are created at a specific point. Giving them an identifiable name makes it easier to remember details about the system state when the checkpoint was created.

By default, the name of a checkpoint is the name of the virtual machine combined with the date and time the checkpoint was taken. This is the standard format:

```
virtual_machine_name (MM/DD/YYY -hh:mm:ss AM\PM)
```

Names are limited to 100 characters, and the name cannot be blank.

Using Hyper-V Manager

1. In **Hyper-V Manager**, select the virtual machine.
2. Right-click the checkpoint, and then select **Rename**.
3. Enter in the new name for the checkpoint. It must be less than 100 characters, and the field cannot be empty.
4. Click **ENTER** when you are done.

Using PowerShell

```PowerShell
Rename-VMCheckpoint -VMName <virtual machine name> -Name <checkpoint name> -NewName <new checkpoint name>
```

Deleting checkpoints

Deleting checkpoints can help create space on your Hyper-V host.

Behind the scenes, checkpoints are stored as .avhdx files in the same location as the .vhdx files for the virtual machine. When you delete a checkpoint, Hyper-V merges the .avhdx and .vhdx files for you. Once completed, the checkpoint's .avhdx file will be deleted from the file system.

You should not delete the .avhdx files directly.

Using Hyper-V Manager

To cleanly delete a checkpoint:

1. In **Hyper-V Manager**, select the virtual machine.
2. In the **Checkpoints** section, right-click the checkpoint that you want to delete and click Delete. You can also delete a checkpoint and all subsequent checkpoints. To do so, right-click the earliest checkpoint that you want to delete, and then click **Delete Checkpoint Subtree**.
3. You might be asked to verify that you want to delete the checkpoint. Confirm that it is the correct checkpoint, and then click **Delete**.

Using PowerShell

```PowerShell
Remove-VMCheckpoint -VMName <virtual machine name> -Name <checkpoint name>
```

Exporting checkpoints

Export bundles the checkpoint as a virtual machine so the checkpoint can be moved to a new location. Once imported, the checkpoint is restored as a virtual machine. Exported checkpoints can be used for backup.

Using PowerShell

```PowerShell
Export-VMCheckpoint -VMName <virtual machine name> -Name <checkpoint name> -Path <path for export>
```

Enable or disable checkpoints

1. In **Hyper-V Manager**, right-click the name of the virtual machine, and click **Settings**.
2. In the **Management** section, select **Checkpoints**.
3. To allow checkpoints to be taken off this virtual machine, make sure Enable Checkpoints is selected -- this is the default behavior.
   To disable checkpoints, deselect the **Enable Checkpoints** check box.
4. Click **Apply** to apply your changes. If you are done, click **OK** to close the dialog box.

Configure checkpoint location

If the virtual machine has no checkpoints, you can change where the checkpoint configuration and saved state files are stored.

1. In **Hyper-V Manager**, right-click the name of the virtual machine, and click **Settings**.
2. In the **Management** section, select **Checkpoints** or **Checkpoint File Location**.
3. In **Checkpoint File Location**, enter the path to the folder where you would like to store the files.
4. Click **Apply** to apply your changes. If you are done, click **OK** to close the dialog box.

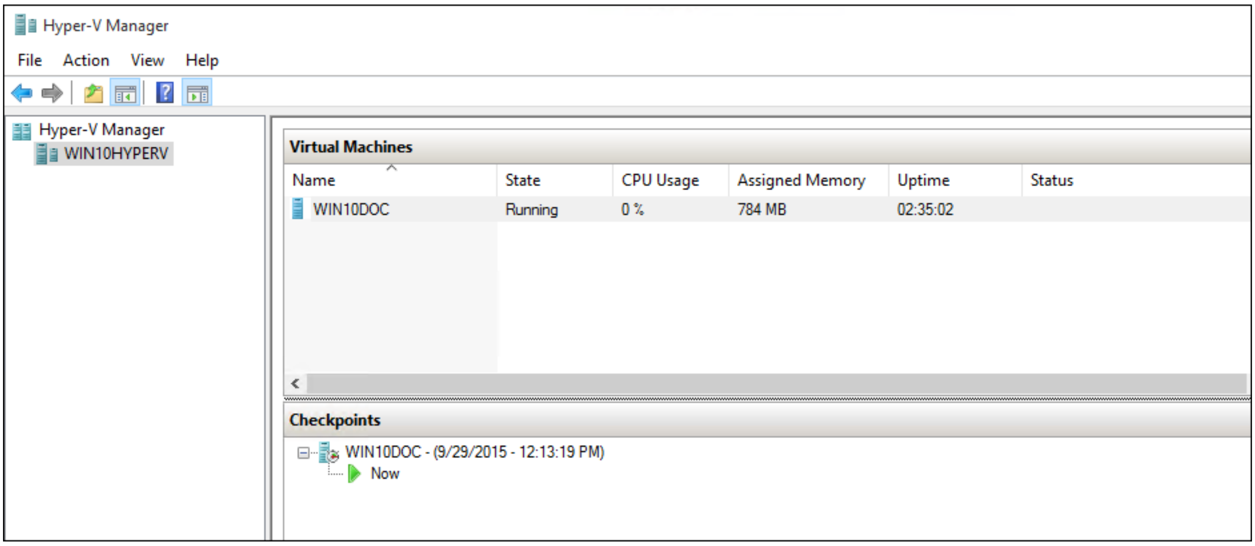
The default location for storing checkpoint configuration files is: `%systemroot%\ProgramData\Microsoft\Windows\Hyper-V\Snapshots`.

Checkpoint demo

This exercise walks through creating and applying a standard checkpoint versus a production checkpoint. For this example, you will make a simple change to the virtual machine and observe the different behavior.

Standard checkpoint

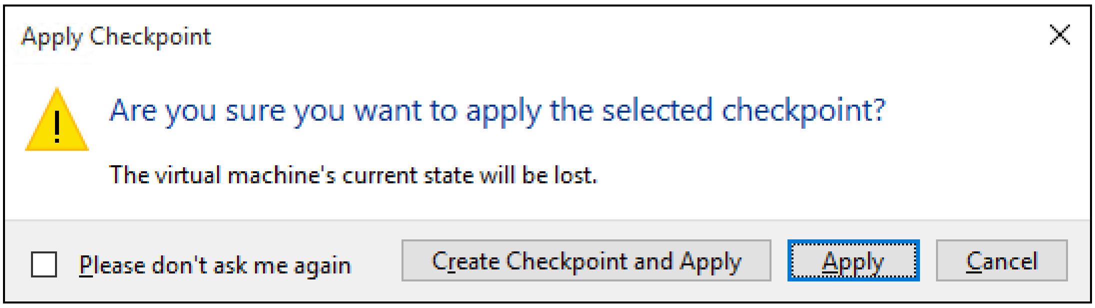
1. Log into your virtual machine and create a text file on the desktop.
2. Open the file with Notepad and enter the text ‘This is a Standard Checkpoint.’ **Do not save the file or close Notepad**.
3. Change the checkpoint to standard -- instructions here.
4. Create a new checkpoint.



Apply the Standard Checkpoint with Hyper-V Manager

Now that a checkpoint exists, make a modification to the virtual machine and then apply the checkpoint to revert the virtual machine back to the saved state.

1. Close the text file if it is still open and delete it from the virtual machine's desktop.
2. Open Hyper-V Manager, right click on the standard checkpoint, and select Apply.
3. Select Apply on the Apply Checkpoint notification window.



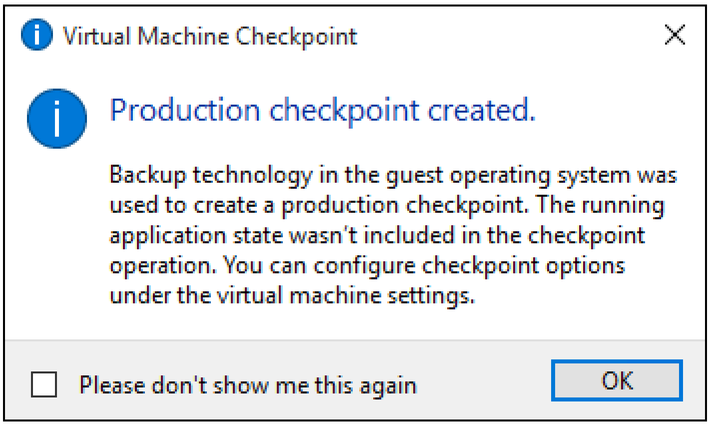
Once the checkpoint has been applied, notice that not only is the text file present, but the system is in the exact state that it was when the checkpoint was created. In this case Notepad is open and the text file loaded.

Production checkpoint

Let’s now examine production checkpoints. This process is almost identical to working with a standard checkpoint, however will have slightly different results. Before beginning make sure you have a virtual machine and that you have changes the checkpoint type to Production checkpoints.

Modify the virtual machine and Create a Production Checkpoint

1. Log into the virtual machine and create a new text file. If you followed the previous exercise, you can use the existing text file.
2. Enter ‘This is a Production Checkpoint.’ into the text file, save the file but **do not close Notepad**.
3. Open Hyper-V Manager, right click on the virtual machine, and select **Checkpoint**.
4. Click **OK** on the Production Checkpoint Created Window.



Apply the Production Checkpoint with Hyper-V Manager

Now that a checkpoint exists make a modification to the system and then apply the checkpoint to revert the virtual machine back to the saved state.

1. Close the text file if it is still open and delete it from the virtual machine's desktop.
2. Open Hyper-V Manager, right click on the production checkpoint, and select **Apply**.
3. Select **Apply** on the Apply Checkpoint notification window.

Once the production checkpoint has been applied, noticed that the virtual machine is in an off state.

1. Start and log into the virtual machine.
2. Take note that the text file has been restored. But unlike the standard checkpoint, Notepad is not open.

Virtual Machine automation and management using PowerShell

Article • 05/09/2023

You can use PowerShell Direct to run arbitrary PowerShell in a Windows 10 or Windows Server 2016 virtual machine from your Hyper-V host regardless of network configuration or remote management settings.

Here are some ways you can run PowerShell Direct:

- As an interactive session using the Enter-PSSession cmdlet
- As a single-use section to execute a single command or script using the Invoke-Command cmdlet
- As a persistant session (build 14280 and later) using the New-PSSession, Copy-Item, and Remove-PSSession cmdlets

Requirements

Operating system requirements:

- Host: Windows 10, Windows Server 2016, or later running Hyper-V.
- Guest/Virtual Machine: Windows 10, Windows Server 2016, or later.

If you're managing older virtual machines, use Virtual Machine Connection (VMConnect) or configure a virtual network for the virtual machine.

Configuration requirements:

- The virtual machine must be running locally on the host.
- The virtual machine must be turned on and running with at least one configured user profile.
- You must be logged into the host computer as a Hyper-V administrator.
- You must supply valid user credentials for the virtual machine.

Create and exit an interactive PowerShell session

The easiest way to run PowerShell commands in a virtual machine is to start an interactive session.

When the session starts, the commands that you type run on the virtual machine, just as though you typed them directly into a PowerShell session on the virtual machine itself.

To start an interactive session:

1. On the Hyper-V host, open PowerShell as Administrator.

2. Run one of the following commands to create an interactive session using the virtual machine name or GUID:

PowerShell

```powershell
Enter-PSSession -VMName <VMName>
Enter-PSSession -VMId <VMId>
```

Provide credentials for the virtual machine when prompted.

3. Run commands on your virtual machine.

You should see the VMName as the prefix for your PowerShell prompt as shown:

```
[VMName]: PS C:\>
```

Any command run will be running on your virtual machine. To test, you can run `ipconfig` or `hostname` to make sure that these commands are running in the virtual machine.

4. When you're done, run the following command to close the session:

   PowerShell

   ```powershell
   Exit-PSSession
   ```

> Note: If your session won't connect, see the troubleshooting for potential causes.

To learn more about these cmdlets, see Enter-PSSession and Exit-PSSession.

Run a script or command with Invoke-Command

PowerShell Direct with Invoke-Command is perfect for situations where you need to run one command or one script on a virtual machine but do not need to continue interacting with the virtual machine beyond that point.

To run a single command:

1. On the Hyper-V host, open PowerShell as Administrator.

2. Run one of the following commands to create a session using the virtual machine name or GUID:

   PowerShell

   ```
   Invoke-Command -VMName <VMName> -ScriptBlock { command }
   Invoke-Command -VMId <VMId> -ScriptBlock { command }
   ```

   Provide credentials for the virtual machine when prompted.

   The command will execute on the virtual machine, if there is output to the console, it'll be printed to your console. The connection will be closed automatically as soon as the command runs.

To run a script:

1. On the Hyper-V host, open PowerShell as Administrator.

2. Run one of the following commands to create a session using the virtual machine name or GUID:

   PowerShell

   ```
   Invoke-Command -VMName <VMName> -FilePath
   C:\host\script_path\script.ps1
   Invoke-Command -VMId <VMId> -FilePath C:\host\script_path\script.ps1
   ```

   Provide credentials for the virtual machine when prompted.

   The script will execute on the virtual machine. The connection will be closed automatically as soon as the command runs.

To learn more about this cmdlet, see Invoke-Command.

Copy files with New-PSSession and Copy-Item

> **Note:** PowerShell Direct only supports persistent sessions in Windows builds 14280 and later

Persistent PowerShell sessions are incredibly useful when writing scripts that coordinate actions across one or more remote machines. Once created, persistent sessions exist in the background until you decide to delete them. This means you can reference the same session over and over again with `Invoke-Command` or `Enter-PSSession` without passing credentials.

By the same token, sessions hold state. Since persistent sessions persist, any variables created in a session or passed to a session will be preserved across multiple calls. There are a number of tools available for working with persistent sessions. For this example, we will use New-PSSession and Copy-Item to move data from the host to a virtual machine and from a virtual machine to the host.

To create a session then copy files:

1. On the Hyper-V host, open PowerShell as Administrator.

2. Run one of the following commands to create a persistent PowerShell session to the virtual machine using `New-PSSession`.

PowerShell

```
$s = New-PSSession -VMName <VMName> -Credential (Get-Credential)
$s = New-PSSession -VMId <VMId> -Credential (Get-Credential)
```

Provide credentials for the virtual machine when prompted.

> **Warning:**
> There is a bug in builds before 14500. If credentials aren't explicitly specified with `-Credential` flag, the service in the guest will crash and will need to be restarted. If you hit this issue, workaround instructions are available here.

3. Copy a file into the virtual machine.

To copy `C:\host_path\data.txt` to the virtual machine from the host machine, run:

PowerShell

```
Copy-Item -ToSession $s -Path C:\host_path\data.txt -Destination
C:\guest_path\
```

4. Copy a file from the virtual machine (on to the host).

To copy `C:\guest_path\data.txt` to the host from the virtual machine, run:

PowerShell

```
Copy-Item -FromSession $s -Path C:\guest_path\data.txt -Destination
C:\host_path\
```

5. Stop the persistent session using `Remove-PSSession`.

PowerShell

```
Remove-PSSession $s
```

Troubleshooting

There are a small set of common error messages surfaced through PowerShell Direct. Here are the most common, some causes, and tools for diagnosing issues.

-VMName or -VMID parameters don't exist

Problem:
`Enter-PSSession`, `Invoke-Command`, or `New-PSSession` do not have a `-VMName` or `-VMId` parameter.

Potential causes:
The most likely issue is that PowerShell Direct isn't supported by your host operating system.

You can check your Windows build by running the following command:

PowerShell

```
[System.Environment]::OSVersion.Version
```

If you are running a supported build, it is also possible your version of PowerShell does not run PowerShell Direct. For PowerShell Direct and JEA, the major version must be 5 or later.

You can check your PowerShell version build by running the following command:

PowerShell

```
$PSVersionTable.PSVersion
```

Error: A remote session might have ended

> **Note:**
> For Enter-PSSession between host builds 10240 and 12400, all errors below reported as "A remote session might have ended".

Error message:

```
Enter-PSSession : An error has occurred which Windows PowerShell cannot
handle. A remote session might have ended.
```

Potential causes:

- The virtual machine exists but is not running.
- The guest OS does not support PowerShell Direct (see requirements)
- PowerShell isn't available in the guest yet
  - The operating system hasn't finished booting
  - The operating system can't boot correctly
  - Some boot time event needs user input

You can use the Get-VM cmdlet to check to see which VMs are running on the host.

Error message:

```
New-PSSession : An error has occurred which Windows PowerShell cannot
handle. A remote session might have ended.
```

Potential causes:

- One of the reasons listed above -- they all are equally applicable to `New-PSSession`
- A bug in current builds where credentials must be explicitly passed with `-Credential`. When this happens, the entire service hangs in the guest operating system and needs to be restarted. You can check if the session is still available with Enter-PSSession.

To work around the credential issue, log into the virtual machine using VMConnect, open PowerShell, and restart the vmicvmsession service using the following PowerShell:

```PowerShell
Restart-Service -Name vmicvmsession
```

Error: Parameter set cannot be resolved

Error message:

```
Enter-PSSession : Parameter set cannot be resolved using the specified named parameters.
```

Potential causes:

- `-RunAsAdministrator` is not supported when connecting to virtual machines.

  When connecting to a Windows container, the `-RunAsAdministrator` flag allows Administrator connections without explicit credentials. Since virtual machines do not give the host implied administrator access, you need to explicitly enter credentials.

Administrator credentials can be passed to the virtual machine with the `-Credential` parameter or by entering them manually when prompted.

Error: The credential is invalid.

Error message:

```
Enter-PSSession : The credential is invalid.
```

Potential causes:

- The guest credentials couldn't be validated
  - The supplied credentials were incorrect.
  - There are no user accounts in the guest (the OS hasn't booted before)
  - If connecting as Administrator: Administrator has not been set as an active user. Learn more here.

Error: The input VMName parameter does not resolve to any virtual machine.

Error message:

```
Enter-PSSession : The input VMName parameter does not resolve to any virtual
machine.
```

Potential causes:

- You are not a Hyper-V Administrator.
- The virtual machine doesn't exist.

You can use the Get-VM cmdlet to check that the credentials you're using have the Hyper-V administrator role and to see which VMs are running locally on the host and booted.

Samples and User Guides

PowerShell Direct supports JEA (Just Enough Administration). Check out this user guide to try it.

Check out samples on GitHub .

Try pre-release features for Hyper-V

Article • 04/26/2022

> This is preliminary content and subject to change.
> Pre-release virtual machines are intended for development or test environments only as they are not supported by Microsoft.

Get early access to pre-release features for Hyper-V on Windows Server 2016 Technical Preview to try out in your development or test environments. You can be the first to see the latest Hyper-V features and help shape the product by providing early feedback.

The virtual machines you create as pre-release have no build-to-build compatibility or future support. Don't use them in a production environment.

Here are some more reasons why these are for non-production environments only:

- There is no forward compatibility for pre-release virtual machines. You can't upgrade these virtual machines to a new configuration version.
- Pre-release virtual machines don't have a consistent definition between builds. If you update the host operating system, existing pre-release virtual machines may be incompatible with the host. These virtual machines may not start, or may initially appear to work but later run into significant compatibility issues.
- If you import a pre-release virtual machine to a host with a different build, the results are unpredictable. You can move a pre-release virtual machine to another host. But this scenario is only expected to work if both hosts are running the same build.

Create a pre-release virtual machine

You can create a pre-release virtual machine on Hyper-V hosts that run Windows Server 2016 Technical Preview.

1. On the Windows desktop, click the Start button and type any part of the name **Windows PowerShell**.
2. Right-click **Windows PowerShell** and select **Run as Administrator**.
3. Use the New-VM cmdlet with the -Prerelease flag to create the pre-release virtual machine. For example, run the following command where VM Name is the name of the virtual machine that you want to create.

PowerShell

```
New-VM -Name <VM Name> -Prerelease
```

Other examples you can use the -Prerelease flag with:

- To create a virtual machine that uses an existing virtual hard disk or a new hard disk, see the PowerShell examples in Create a virtual machine in Hyper-V on Windows Server 2016 Technical Preview.
- To create a new virtual hard disk that boots to an operating system image, see the PowerShell example in Deploy a Windows Virtual Machine in Hyper-V on Windows 10.

The examples covered in those articles work for Hyper-V hosts that run Windows 10 or Windows Server 2016 Technical Preview. But right now, you can only use the -Prerelease flag to create a pre-release virtual machine on Hyper-V hosts that run Windows Server 2016 Technical Preview.

See also

- Virtualization Blog - Learn about the pre-release features that are available and how to try them out.
- Supported virtual machine configuration versions - Learn how to check the virtual machine configuration version and which versions are supported by Microsoft.

What is Nested Virtualization?

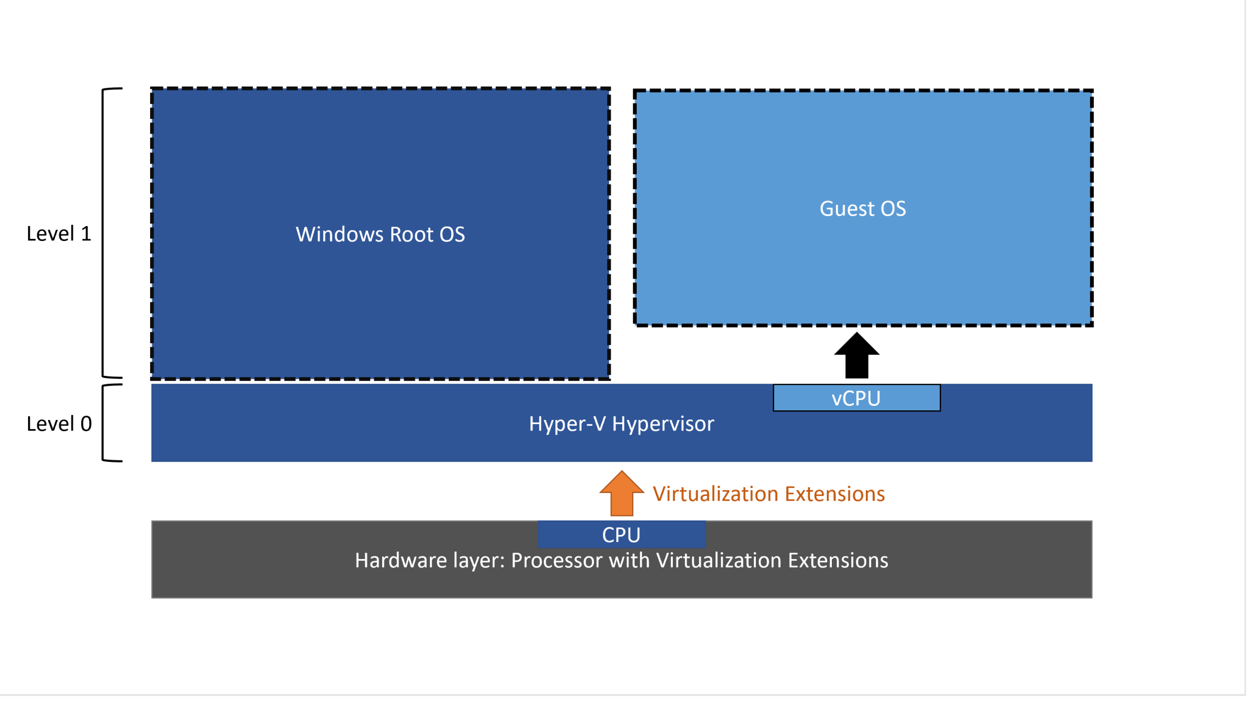
Article • 07/27/2023

Nested virtualization is a feature that lets you run Hyper-V inside a Hyper-V virtual machine (VM). Over the years hardware has improved and the use cases for Nested Virtualization have expanded. For example, Nested Virtualization can be useful for:

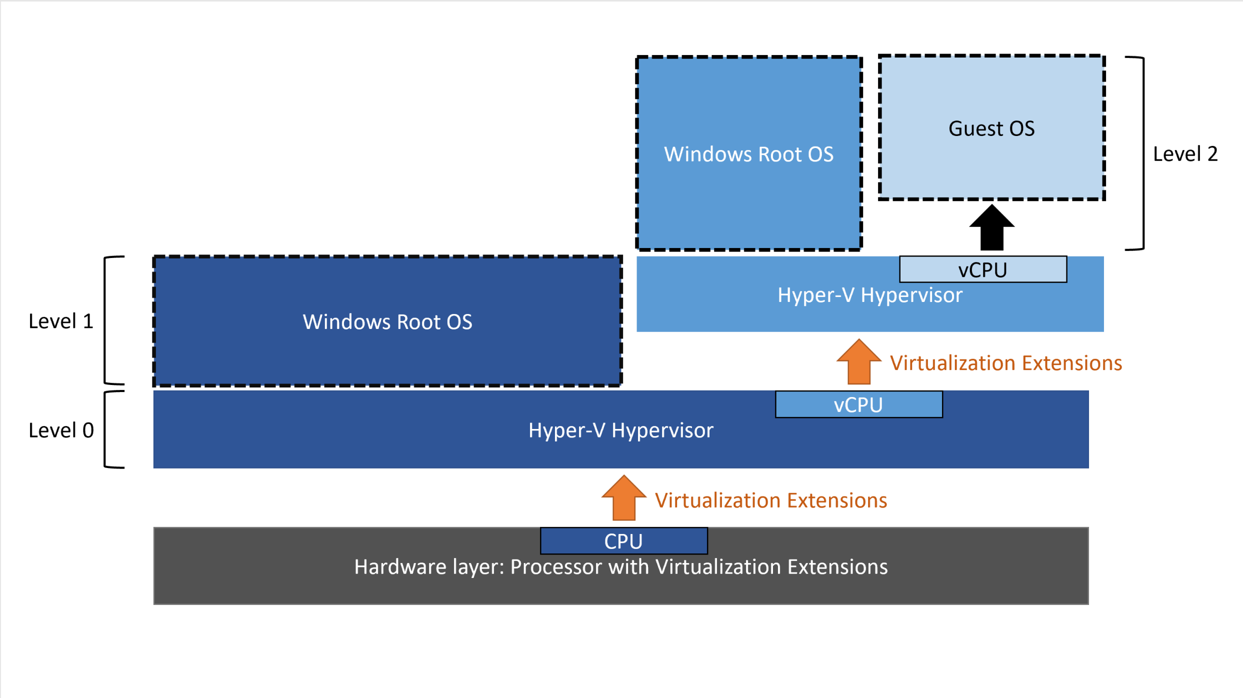
- Running applications or emulators in a nested VM
- Testing software releases on VMs
- Reducing deployment times for training environments
- Using Hyper-V isolation for containers

Modern processors include hardware features that make virtualization faster and more secure. Hyper-V relies on these processor extensions to run virtual machines, for example, Intel VT-x and AMD-V. Nested virtualization makes this hardware support available to guest virtual machines.

The following diagram shows Hyper-V without nesting. The Hyper-V hypervisor takes full control of the hardware virtualization capabilities (orange arrow), and doesn't expose them to the guest operating system.



In contrast, the following diagram shows Hyper-V with Nested Virtualization enabled. In this case, Hyper-V exposes the hardware virtualization extensions to its virtual machines. With nesting enabled, a guest virtual machine can install its own hypervisor and run its own guest VMs.



Dynamic Memory and Runtime Memory Resize

When Hyper-V is running inside a virtual machine, the virtual machine must be turned off to adjust its memory. Meaning that even if dynamic memory is enabled, the amount of memory doesn't fluctuate. Simply enabling nested virtualization has no effect on dynamic memory or runtime memory resize.

For virtual machines without dynamic memory enabled, attempting to adjust the amount of memory while running fails. The incompatibility only occurs while Hyper-V is running in the VM.

Third party virtualization apps

Virtualization applications other than Hyper-V aren't supported in Hyper-V virtual machines, and are likely to fail. Virtualization applications include any software that requires hardware virtualization extensions.

Supported scenarios

Using a nested Hyper-V VM in production is supported for both Azure and on-premises in the following scenarios. We also recommend you make sure that your services and applications are also supported.

Nested Virtualization isn't suitable for Windows Server Failover Clustering, and performance sensitive applications. We recommended you fully evaluate the services

and applications.

Hyper-V VMs on Hyper-V VMs

Running Hyper-V VMs nested on Hyper-V VMs is great for test labs and evaluation environments. Especially where configurations can be easily modified and saved states can be used to revert to specific configurations. Test labs don't typically require the same service level agreement (SLA) as production environments.

Production environments running Hyper-V VMs running on Hyper-V VMs are supported. However, it's recommended to make sure that your services and applications are also supported. If you use nested Hyper-V VM in production, it is recommended to fully evaluate whether the services or applications provide the expected behavior.

To learn more about setting up Nested Virtualization on Azure, see our Tech Community blog How to Setup Nested Virtualization for Azure VM/VHD.

Third party virtualization on Hyper-V virtualization

Whilst it might be possible for third party virtualization to run on Hyper-V, Microsoft doesn't test this scenario. Third party virtualization on Hyper-V virtualization isn't supported, ensure your hypervisor vendor supports this scenario.

Hyper-V virtualization on third party virtualization

Whilst it might be possible for Hyper-V virtualization to run on third party virtualization, Microsoft doesn't test this scenario. Hyper-V virtualization on third party virtualization isn't supported, ensure your hypervisor vendor supports this scenario.

Azure Stack HCI nested on Hyper-V VMs

Azure Stack HCI is designed and tested to run on validated physical hardware. Azure Stack HCI can run nested in a virtual machine for evaluation, but production environments in a nested configuration aren't supported.

To learn more about Azure Stack HCI nested on Hyper-V VMs, see Nested virtualization in Azure Stack HCI.

Hyper-V isolated containers running nested on Hyper-V

Microsoft offers Hyper-V isolation for containers. This isolation mode offers enhanced security and broader compatibility between host and container versions. With Hyper-V isolation, multiple container instances run concurrently on a host. Each container runs inside of a highly optimized virtual machine and effectively gets its own kernel. Since a Hyper-V isolated container offers isolation through a hypervisor layer between itself and the container host, when the container host is a Hyper-V based virtual machine, there's performance overhead. The associated performance overhead occurs in terms of container start-up time, storage, network, and CPU operations.

When a Hyper-V isolated container is run in a Hyper-V VM, it's running nested. Using a Hyper-V VM opens many useful scenarios but also increases latency, as there are two levels of hypervisors running above the physical host.

Running Hyper-V isolated containers nested on Hyper-V is supported.

To learn more about Nested Hyper-V Containers, see Performance tuning Windows Server Containers.

Running WSL2 in a Hyper-V VM running nested on Hyper-V

Windows Subsystem for Linux (WSL) is a feature of the Windows operating system that enables you to run a Linux file system, along with Linux command-line tools and GUI apps, directly on Windows.

Running WSL2 in a Hyper-V VM running nested on Hyper-V is supported.

To learn more about how to enabled WSL 2 to run in a VM, see Frequently Asked Questions about Windows Subsystem for Linux.

Next step

- Run Hyper-V in a Virtual Machine with Nested Virtualization

Run Hyper-V in a Virtual Machine with Nested Virtualization

Article • 07/27/2023

Nested Virtualization is a feature that allows you to run Hyper-V inside of a Hyper-V virtual machine (VM). Nested Virtualization is helpful for running a Visual Studio phone emulator in a virtual machine, or testing configurations that ordinarily require several hosts.

To learn more about Nested Virtualization and supported scenarios, see What is Nested Virtualization for Hyper-V?.

Prerequisites

Intel processor with VT-x and EPT technology

- The Hyper-V host must be either Windows Server 2016 or later, or Windows 10 or later.
- VM configuration version 8.0 or higher.

AMD EPYC / Ryzen processor or later

- The Hyper-V host must be either Windows Server 2022 or later, or Windows 11 or later.
- VM configuration version 9.3 or higher.

> ⓘ **Note**
>
> The guest can be any Windows supported guest operating system. Newer Windows operating systems may support enlightenments that improve performance.

Configure Nested Virtualization

1. Create a virtual machine. See the prerequisites for the required OS and VM versions.

2. While the virtual machine is in the OFF state, run the following command on the physical Hyper-V host to enable nested virtualization for the virtual machine.

PowerShell

```
Set-VMProcessor -VMName <VMName> -ExposeVirtualizationExtensions $true
```

3. Start the virtual machine.

4. Install Hyper-V within the virtual machine, just like you would for a physical server. For more information on installing Hyper-V, see, Install Hyper-V.

> ⓘ **Note**
>
> When using Windows Server 2019 as the first level VM, the number of vCPUs should be 225 or less.

Disable Nested Virtualization

You can disable nested virtualization for a stopped virtual machine using the following PowerShell command:

PowerShell

```
Set-VMProcessor -VMName <VMName> -ExposeVirtualizationExtensions $false
```

Networking options

There are two options for networking with nested virtual machines:

1. MAC address spoofing
2. NAT networking

MAC address spoofing

In order for network packets to be routed through two virtual switches, MAC address spoofing must be enabled on the first (L1) level of virtual switch. To enable MAC address spoofing, run the following PowerShell command.

PowerShell

```
Get-VMNetworkAdapter -VMName <VMName> | Set-VMNetworkAdapter -
MacAddressSpoofing On
```

Network Address Translation (NAT)

The second option relies on network address translation (NAT). This approach is best suited for cases where MAC address spoofing isn't possible, like in a public cloud environment.

First, a virtual NAT switch must be created in the host virtual machine (the "middle" VM). The following example creates a new internal switch named `VmNAT` and creates a NAT object for all IP addresses in the `192.168.100.0/24` subnet.

PowerShell

```
New-VMSwitch -Name VmNAT -SwitchType Internal
New-NetNat –Name LocalNAT –InternalIPInterfaceAddressPrefix
“192.168.100.0/24”
```

Next, assign an IP address to the net adapter:

PowerShell

```
Get-NetAdapter "vEthernet (VmNat)" | New-NetIPAddress -IPAddress
192.168.100.1 -AddressFamily IPv4 -PrefixLength 24
```

Each nested virtual machine must have an IP address and gateway assigned to it. The gateway IP must point to the NAT adapter from the previous step. You may also want to assign a DNS server:

PowerShell

```
Get-NetAdapter "vEthernet (VmNat)" | New-NetIPAddress -IPAddress
192.168.100.2 -DefaultGateway 192.168.100.1 -AddressFamily IPv4 -
PrefixLength 24
Netsh interface ip add dnsserver “vEthernet (VmNat)” address=<my DNS server>
```

Next steps

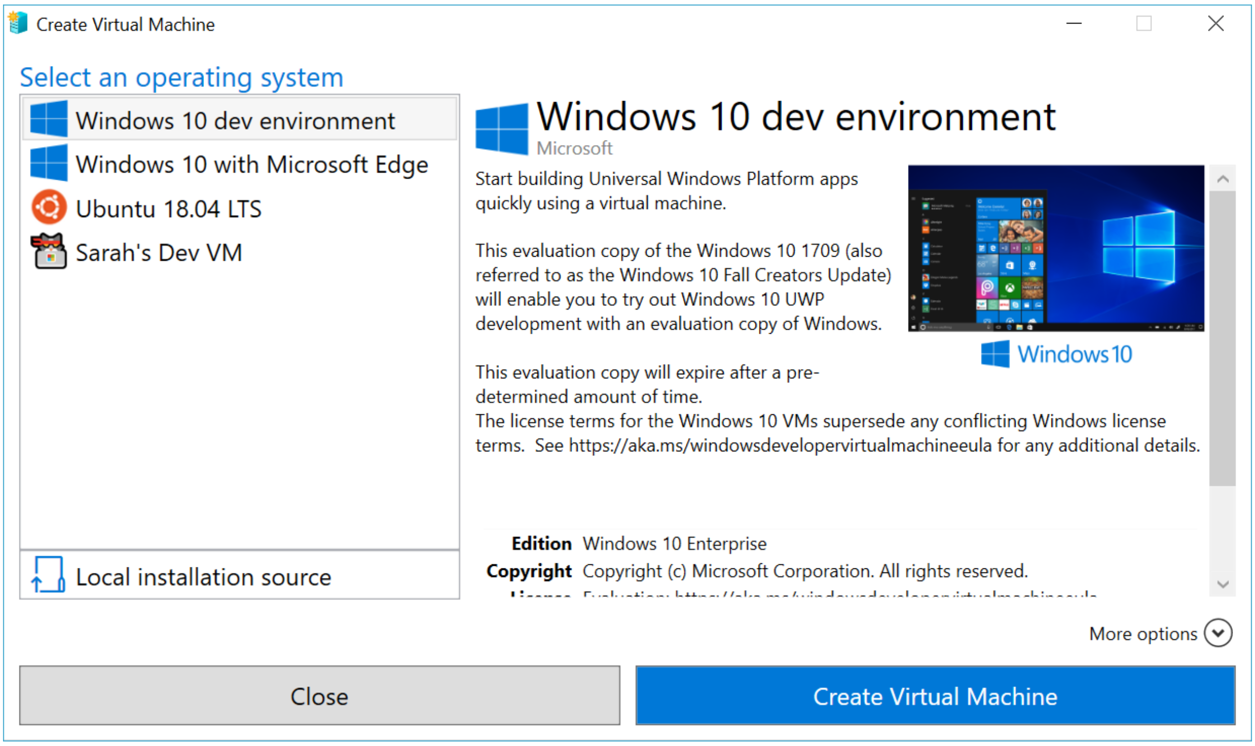
- Remotely manage Hyper-V hosts with Hyper-V Manager

Create a custom virtual machine gallery

Article • 04/26/2022

> Windows 10 Fall Creators Update and later.

In Fall Creators Update, Quick Create expanded to include a virtual machine gallery.



While there is a set of images provided by Microsoft and Microsoft partners, the gallery can also list your own images.

This article details:

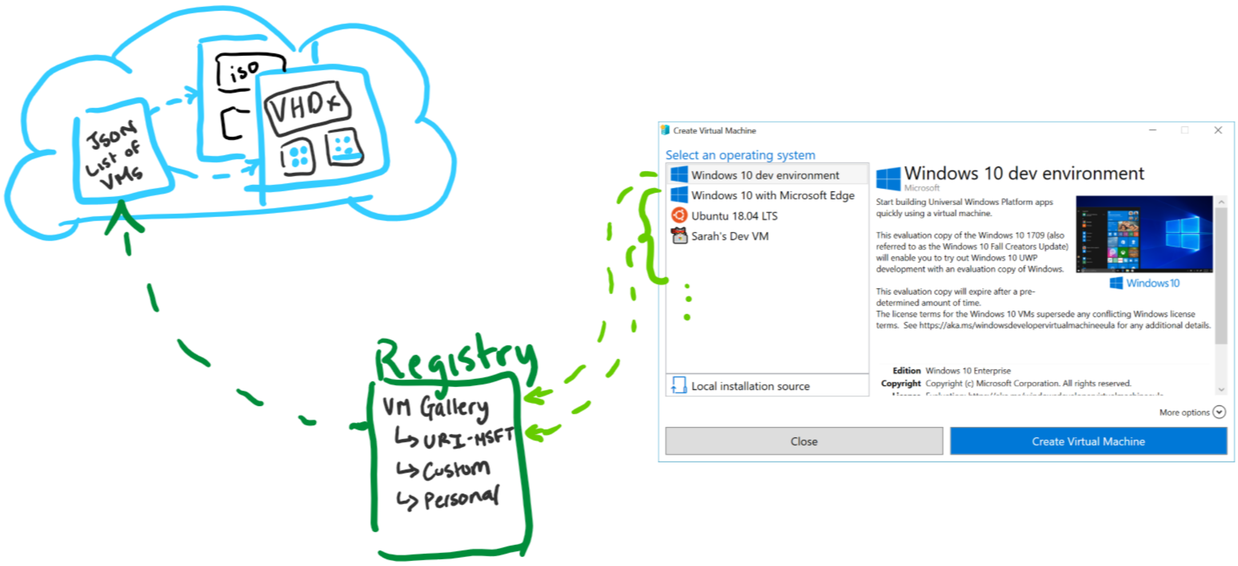
- building virtual machines that are compatible with the gallery.
- creating a new gallery source.
- adding your custom gallery source to the gallery.

Gallery architecture

The virtual machine gallery is a graphical view for a set of virtual machine sources defined in the Windows registry. Each virtual machine source is a path (local path or URI) to a JSON file with virtual machines as list items.

The list of virtual machines you see in the gallery is the full contents of the first source, followed by the contents of the second source, so on and so forth until all of the

available virtual machines have been listed. The list is dynamically created every time you launch the gallery.



Registry Key: `Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Virtualization`

Value Name: `GalleryLocations`

Type: `REG_MULTI_SZ`

Create gallery-compatible virtual machines

Virtual machines in the gallery can be either a disk image (.iso) or virtual hard drive (.vhdx).

Virtual machines made from a virtual hard drive have a few configuration requirements:

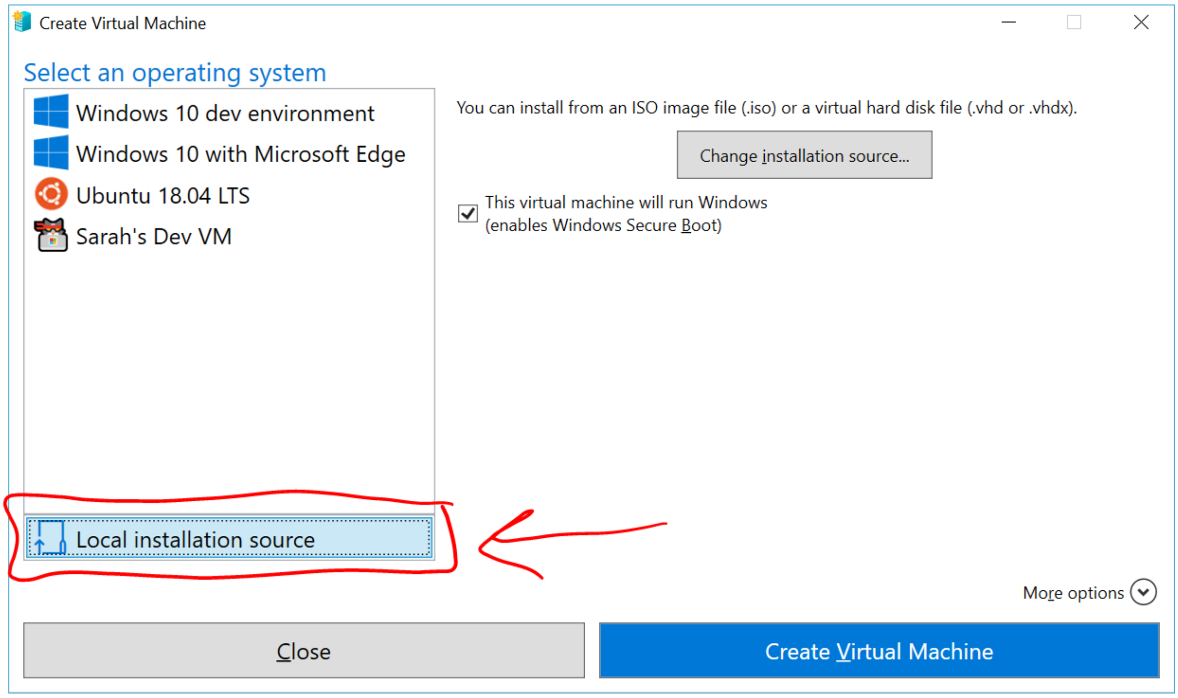
1. Built to support UEFI firmware. If they're created using Hyper-V, that's a Generation 2 VM.
2. The virtual hard drive should be at least 20GB - keep in mind, that's the max size. Hyper-V will not take space the VM isn't actively using.

Testing a new VM image

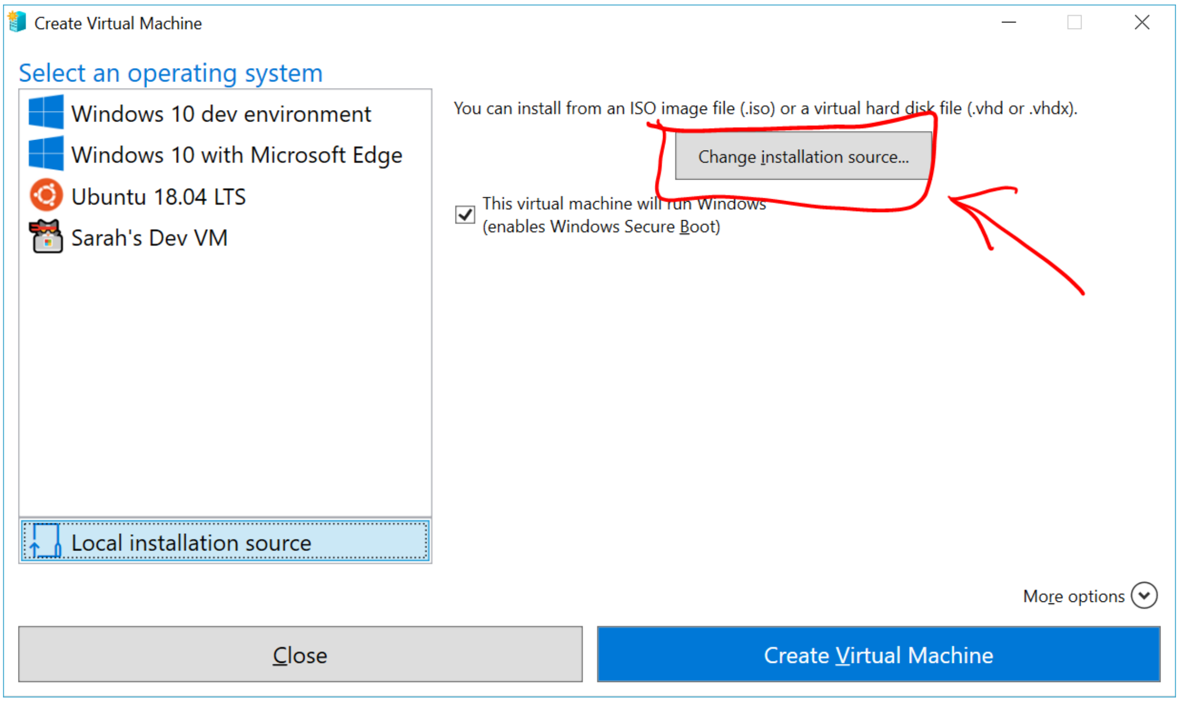
The virtual machine gallery creates virtual machines using the same mechanism as installing from a local installation source.

To validate a virtual machine image will boot and run:

1. Open the VM Gallery (Hyper-V Quick Create) and select **Local Installation Source**.

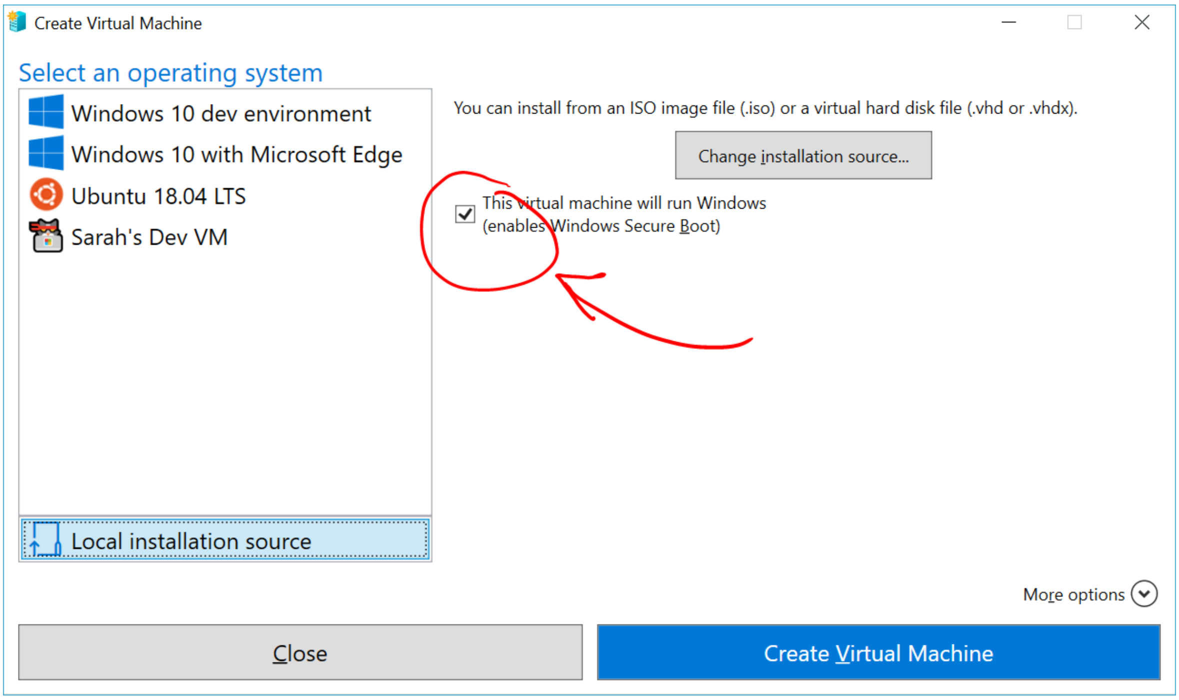


2. Select **Change Installation Source**.



3. Pick the .iso or .vhdx that will be used in the gallery.

4. If the image is a Linux image, deselect the Secure Boot option.

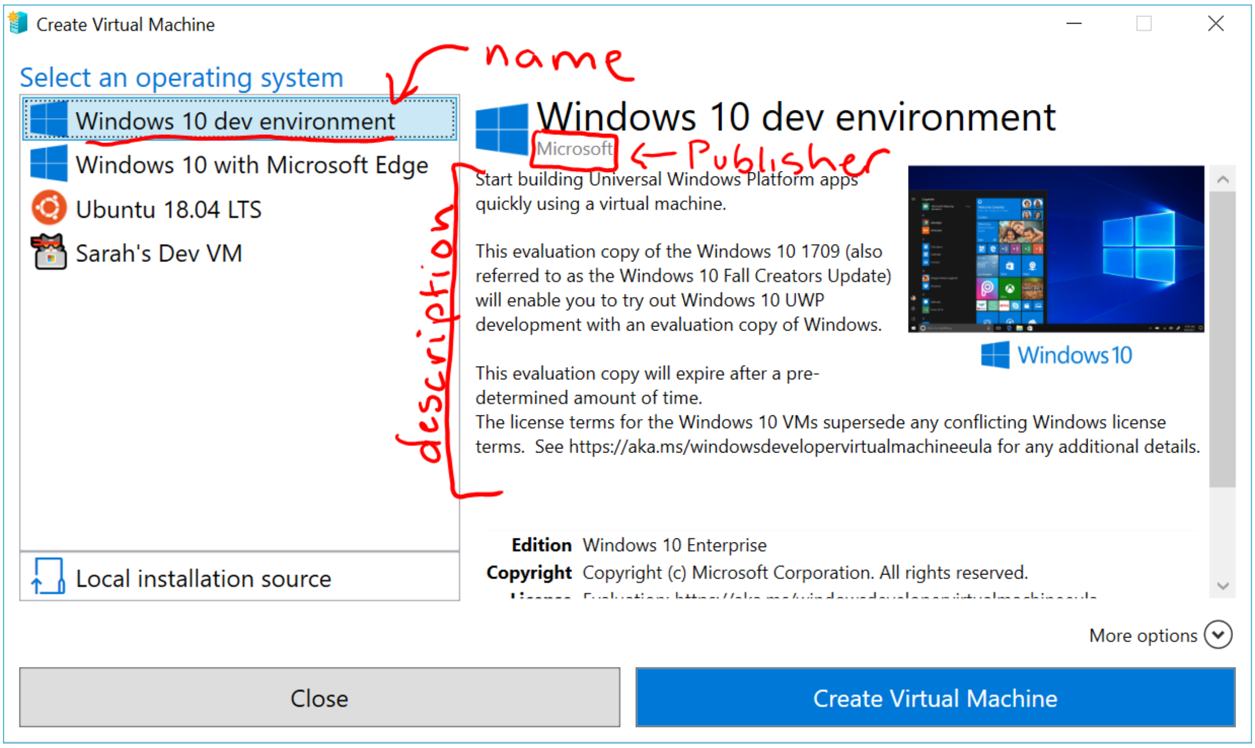


5. Create Virtual Machine. If the virtual machine boots correctly, it's ready for the gallery.

Build a new gallery source

The next step is to create a new gallery source. This is the JSON file that lists your virtual machines and adds all of the extra information you see in the gallery.

Text information:



- **name** - required - this is the name that appears in the left column and also at the top of the virtual machine view.
- **publisher** - required
- **description** - required - List of strings that describe the VM.
- **version** - required
- lastUpdated - defaults to Monday, January 1, 0001.

  The format should be: yyyy-mm-ddThh:mm:ssZ

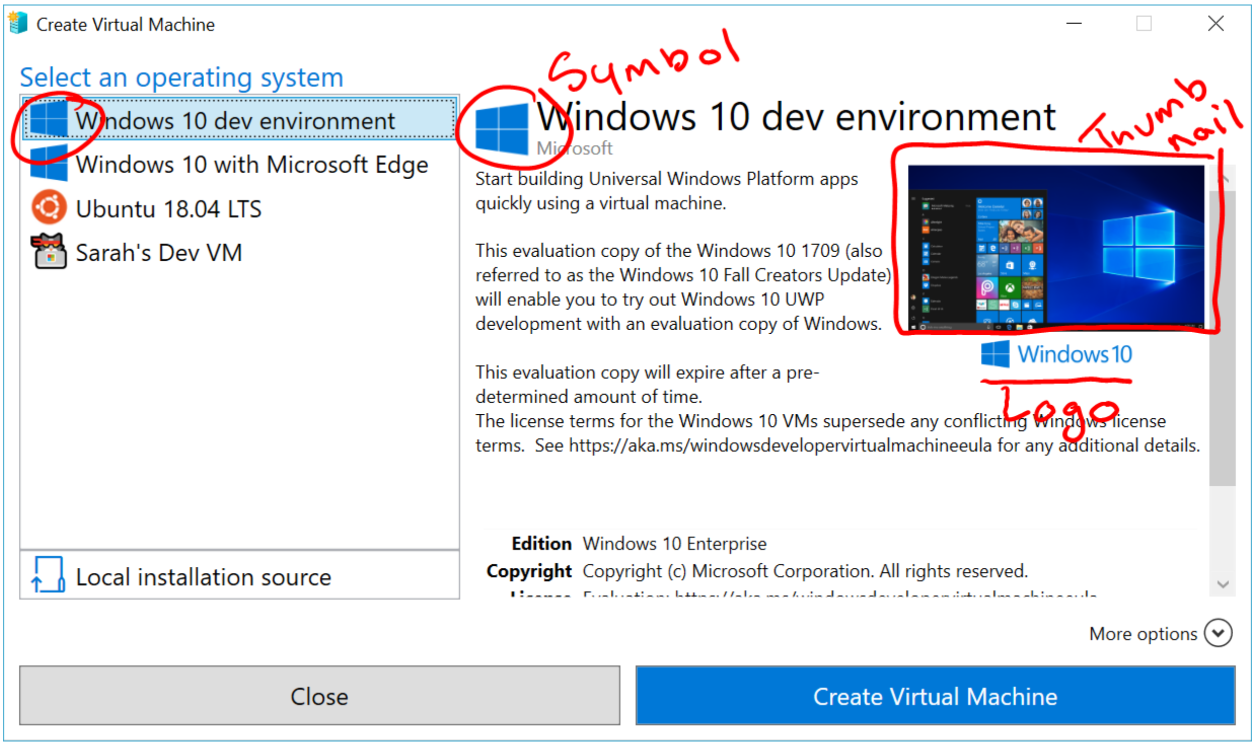
  The following PowerShell command will provide today's date in the proper format and put it on the clipboard:

  PowerShell

  ```
  Get-Date -UFormat "%Y-%m-%dT%TZ" | clip.exe
  ```

- locale - defaults to blank.

Pictures:



- **logo** - required
- symbol
- thumbnail

And, of course, your virtual machine (.iso or .vhdx).

To generate the hashes, you can use the following powershell command:

```PowerShell
Get-FileHash -Path .\TMLogo.jpg -Algorithm SHA256
```

The below JSON template has starter items and the gallery's schema. If you edit it in VSCode, it will automatically provide IntelliSense.

```JSON
{
    "$schema":
"https://raw.githubusercontent.com/MicrosoftDocs/Virtualization-Documentation/live/hyperv-tools/vmgallery/vm-gallery-schema.json",

    "images": [
        {
            "name":  "",
            "version":  "",
            "locale":  "",
            "publisher":  "",
            "lastUpdated":  "",
            "description":  [
                ""
            ],
            "disk":  {
                "uri":  "",
                "hash":  ""
            },
            "logo":  {
                "uri":  "",
                "hash":  ""
            },
            "symbol":  {
                "uri":  "",
                "hash":  ""
            },
            "thumbnail":  {
                "uri":  "",
                "hash":  ""
            }
        }
    ]
}
```

Connect your gallery to the VM Gallery UI

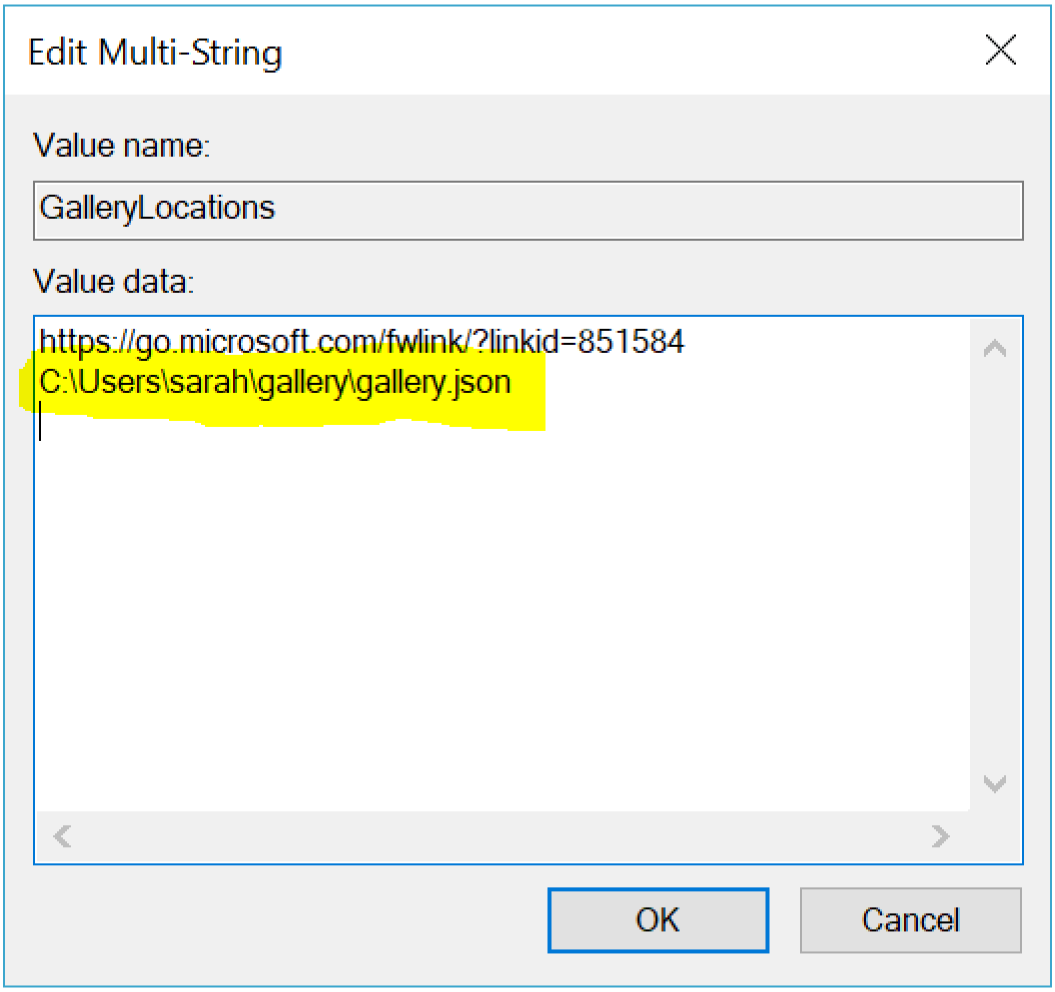
The easiest way to add your custom gallery source to the VM Gallery is to add it in regedit.

1. Open **regedit.exe**
2. Navigate to `Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Virtualization\`
3. Look for the `GalleryLocations` item.

   If it already exists, go to the **Edit** menu and **modify**.

   If it doesn't already exist, go to the **Edit** menu, navigate through **New** to **Multi-String Value**

4. Add your gallery to the `GalleryLocations` registry key.



Troubleshooting

Check for errors loading Gallery

The virtual machine gallery does provide error reporting in the Windows Event Viewer. To check for errors:

1. Open Event Viewer
2. Navigate to **Windows Logs** -> **Application**
3. Look for events from Source VMCreate.

Resources

There are a handfull of gallery scripts and helpers in GitHub link.

See a sample gallery entry here. This is the JSON file that defines the in-box gallery.

Set up a NAT network

Article • 01/17/2023

Windows 10 Hyper-V allows native network address translation (NAT) for a virtual network.

This guide will walk you through:

- creating a NAT network
- connecting an existing virtual machine to your new network
- confirming that the virtual machine is connected correctly

Requirements:

- Windows 10 Anniversary Update or later
- Hyper-V is enabled (instructions here)

> **Note:** Currently, you are limited to one NAT network per host. For additional details on the Windows NAT (WinNAT) implementation, capabilities, and limitations, please reference the WinNAT capabilities and limitations blog

NAT Overview

NAT gives a virtual machine access to network resources using the host computer's IP address and a port through an internal Hyper-V Virtual Switch.

Network Address Translation (NAT) is a networking mode designed to conserve IP addresses by mapping an external IP address and port to a much larger set of internal IP addresses. Basically, a NAT uses a flow table to route traffic from an external (host) IP Address and port number to the correct internal IP address associated with an endpoint on the network (virtual machine, computer, container, etc.)

Additionally, NAT allows multiple virtual machines to host applications that require identical (internal) communication ports by mapping these to unique external ports.

For all of these reasons, NAT networking is very common for container technology (see Container Networking).

Create a NAT virtual network

Let's walk through setting up a new NAT network.

1. Open a PowerShell console as Administrator.

2. Create an internal switch.

PowerShell

```
New-VMSwitch -SwitchName "SwitchName" -SwitchType Internal
```

3. Find the interface index of the virtual switch you just created.

   You can find the interface index by running `Get-NetAdapter`

   Your output should look something like this:

Console

```
PS C:\> Get-NetAdapter

Name                      InterfaceDescription                  ifIndex Status
MacAddress               LinkSpeed
----                      --------------------                  ------- ------
----------               ---------
vEthernet (intSwitch) Hyper-V Virtual Ethernet Adapter          24 Up
00-15-5D-00-6A-01         10 Gbps
Wi-Fi                     Marvell AVASTAR Wireless-AC Net...      18 Up
98-5F-D3-34-0C-D3        300 Mbps
Bluetooth Network ... Bluetooth Device (Personal Area...      21
Disconnected 98-5F-D3-34-0C-D4        3 Mbps
```

   The internal switch will have a name like `vEthernet (SwitchName)` and an Interface Description of `Hyper-V Virtual Ethernet Adapter`. Take note of its `ifIndex` to use in the next step.

4. Configure the NAT gateway using New-NetIPAddress.

   Here is the generic command:

PowerShell

```
New-NetIPAddress -IPAddress <NAT Gateway IP> -PrefixLength <NAT Subnet
Prefix Length> -InterfaceIndex <ifIndex>
```

   In order to configure the gateway, you'll need a bit of information about your network:

- **IPAddress** -- NAT Gateway IP specifies the IPv4 or IPv6 address to use as the NAT gateway IP. The generic form will be a.b.c.1 (e.g. 172.16.0.1). While the final position doesn't have to be .1, it usually is (based on prefix length). This IP address is in the range of addresses used by the guest virtual machines. For example if the guest VMs use IP range 172.16.0.0, then you can use an IP address 172.16.0.100 as the NAT Gateway.

  A common gateway IP is 192.168.0.1

- **PrefixLength** -- NAT Subnet Prefix Length defines the NAT local subnet size (subnet mask). The subnet prefix length will be an integer value between 0 and 32.

  0 would map the entire internet, 32 would only allow one mapped IP. Common values range from 24 to 12 depending on how many IPs need to be attached to the NAT.

  A common PrefixLength is 24 -- this is a subnet mask of 255.255.255.0

- **InterfaceIndex** -- ifIndex is the interface index of the virtual switch, which you determined in the previous step.

Run the following to create the NAT Gateway:

PowerShell

```
New-NetIPAddress -IPAddress 192.168.0.1 -PrefixLength 24 -
InterfaceIndex 24
```

5. Configure the NAT network using New-NetNat.

   Here is the generic command:

PowerShell

```
New-NetNat -Name <NATOutsideName> -InternalIPInterfaceAddressPrefix
<NAT subnet prefix>
```

In order to configure the gateway, you'll need to provide information about the network and NAT Gateway:

- **Name** -- NATOutsideName describes the name of the NAT network. You'll use this to remove the NAT network.

- **InternalIPInterfaceAddressPrefix** -- NAT subnet prefix describes both the NAT Gateway IP prefix from above as well as the NAT Subnet Prefix Length from above.

The generic form will be a.b.c.0/NAT Subnet Prefix Length

From the above, for this example, we'll use 192.168.0.0/24

For our example, run the following to setup the NAT network:

```PowerShell
New-NetNat -Name MyNATnetwork -InternalIPInterfaceAddressPrefix
192.168.0.0/24
```

Congratulations! You now have a virtual NAT network! To add a virtual machine, to the NAT network follow these instructions.

Connect a virtual machine

To connect a virtual machine to your new NAT network, connect the internal switch you created in the first step of the NAT Network Setup section to your virtual machine using the VM Settings menu.

Since WinNAT by itself does not allocate and assign IP addresses to an endpoint (e.g. VM), you will need to do this manually from within the VM itself - i.e. set IP address within range of NAT internal prefix, set default gateway IP address, set DNS server information. The only caveat to this is when the endpoint is attached to a container. In this case, the Host Network Service (HNS) allocates and uses the Host Compute Service (HCS) to assign the IP address, gateway IP, and DNS info to the container directly.

Configuration Example: Attaching VMs and Containers to a NAT network

If you need to attach multiple VMs and containers to a single NAT, you will need to ensure that the NAT internal subnet prefix is large enough to encompass the IP ranges being assigned by different applications or services (e.g. Docker for Windows and Windows Container – HNS). This will require either application-level assignment of IPs and network configuration or manual configuration which must be done by an admin and guaranteed not to re-use existing IP assignments on the same host.

Docker for Windows (Linux VM) and Windows Containers

The solution below will allow both Docker for Windows (Linux VM running Linux containers) and Windows Containers to share the same WinNAT instance using separate internal vSwitches. Connectivity between both Linux and Windows containers will work.

User has connected VMs to a NAT network through an internal vSwitch named “VMNAT” and now wants to install Windows Container feature with docker engine

```
PS C:\> Get-NetNat “VMNAT”| Remove-NetNat (this will remove the NAT but keep
the internal vSwitch).
Install Windows Container Feature
DO NOT START Docker Service (daemon)
Edit the arguments passed to the docker daemon (dockerd) by adding –fixed-
cidr=<container prefix> parameter. This tells docker to create a default nat
network with the IP subnet <container prefix> (e.g. 192.168.1.0/24) so that
HNS can allocate IPs from this prefix.
PS C:\> Start-Service Docker; Stop-Service Docker
PS C:\> Get-NetNat | Remove-NetNAT (again, this will remove the NAT but keep
the internal vSwitch)
PS C:\> New-NetNat -Name SharedNAT -InternalIPInterfaceAddressPrefix <shared
prefix>
PS C:\> Start-Service docker
```

Docker/HNS will assign IPs to Windows containers and Admin will assign IPs to VMs from the difference set of the two.

User has installed Windows Container feature with docker engine running and now wants to connect VMs to the NAT network

```
PS C:\> Stop-Service docker
PS C:\> Get-ContainerNetwork | Remove-ContainerNetwork -force
PS C:\> Get-NetNat | Remove-NetNat (this will remove the NAT but keep the
internal vSwitch)
Edit the arguments passed to the docker daemon (dockerd) by adding -b “none”
option to the end of docker daemon (dockerd) command to tell docker not to
create a default NAT network.
PS C:\> New-ContainerNetwork –name nat –Mode NAT –subnetprefix <container
prefix> (create a new NAT and internal vSwitch – HNS will allocate IPs to
container endpoints attached to this network from the <container prefix>)
PS C:\> Get-Netnat | Remove-NetNAT (again, this will remove the NAT but keep
the internal vSwitch)
PS C:\> New-NetNat -Name SharedNAT -InternalIPInterfaceAddressPrefix <shared
prefix>
PS C:\> New-VirtualSwitch -Type internal (attach VMs to this new vSwitch)
PS C:\> Start-Service docker
```

Docker/HNS will assign IPs to Windows containers and Admin will assign IPs to VMs from the difference set of the two.

In the end, you should have two internal VM switches and one NetNat shared between them.

Multiple Applications using the same NAT

Some scenarios require multiple applications or services to use the same NAT. In this case, the following workflow must be followed so that multiple applications / services can use a larger NAT internal subnet prefix

We will detail the Docker 4 Windows - Docker Beta - Linux VM co-existing with the Windows Container feature on the same host as an example. This workflow is subject to change

1. C:> net stop docker

2. Stop Docker4Windows MobyLinux VM

3. PS C:> Get-ContainerNetwork | Remove-ContainerNetwork -force

4. PS C:> Get-NetNat | Remove-NetNat
   Removes any previously existing container networks (i.e. deletes vSwitch, deletes NetNat, cleans up)

5. New-ContainerNetwork -Name nat -Mode NAT –subnetprefix 10.0.76.0/24 (this subnet will be used for Windows containers feature) *Creates internal vSwitch named nat*
   Creates NAT network named "nat" with IP prefix 10.0.76.0/24

6. Remove-NetNAT
   Removes both DockerNAT and nat NAT networks (keeps internal vSwitches)

7. New-NetNat -Name DockerNAT -InternalIPInterfaceAddressPrefix 10.0.0.0/17 (this will create a larger NAT network for both D4W and containers to share)
   Creates NAT network named DockerNAT with larger prefix 10.0.0.0/17

8. Run Docker4Windows (MobyLinux.ps1)
   Creates internal vSwitch DockerNAT
   Creates NAT network named "DockerNAT" with IP prefix 10.0.75.0/24

9. Net start docker
   Docker will use the user-defined NAT network as the default to connect Windows containers

In the end, you should have two internal vSwitches – one named DockerNAT and the other named nat. You will only have one NAT network (10.0.0.0/17) confirmed by running Get-NetNat. IP addresses for Windows containers will be assigned by the Windows Host Network Service (HNS) from the 10.0.76.0/24 subnet. Based on the existing MobyLinux.ps1 script, IP addresses for Docker 4 Windows will be assigned from the 10.0.75.0/24 subnet.

Troubleshooting

Multiple NAT networks are not supported

This guide assumes that there are no other NATs on the host. However, applications or services will require the use of a NAT and may create one as part of setup. Since Windows (WinNAT) only supports one internal NAT subnet prefix, trying to create multiple NATs will place the system into an unknown state.

To see if this may be the problem, make sure you only have one NAT:

PowerShell

```
Get-NetNat
```

If a NAT already exists, delete it

PowerShell

```
Get-NetNat | Remove-NetNat
```

Make sure you only have one "internal" vmSwitch for the application or feature (e.g. Windows containers). Record the name of the vSwitch

PowerShell

```
Get-VMSwitch
```

Check to see if there are private IP addresses (e.g. NAT default Gateway IP Address – usually *x.y.z.*1) from the old NAT still assigned to an adapter

PowerShell

```
Get-NetIPAddress -InterfaceAlias "vEthernet (<name of vSwitch>)"
```

If an old private IP address is in use, please delete it

PowerShell

```
Remove-NetIPAddress -InterfaceAlias "vEthernet (<name of vSwitch>)" -
IPAddress <IPAddress>
```

Removing Multiple NATs

We have seen reports of multiple NAT networks created inadvertently. This is due to a bug in recent builds (including Windows Server 2016 Technical Preview 5 and Windows 10 Insider Preview builds). If you see multiple NAT networks, after running docker network ls or Get-ContainerNetwork, please perform the following from an elevated PowerShell:

PowerShell

```
$keys = Get-ChildItem
"HKLM:\SYSTEM\CurrentControlSet\Services\vmsmp\parameters\SwitchList"
foreach($key in $keys)
{
   if ($key.GetValue("FriendlyName") -eq 'nat')
   {
      $newKeyPath = $KeyPath+"\"+$key.PSChildName
      Remove-Item -Path $newKeyPath -Recurse
   }
}
Remove-NetNat -Confirm:$false
Get-ContainerNetwork | Remove-ContainerNetwork
Get-VmSwitch -Name nat | Remove-VmSwitch # failure is expected
Stop-Service docker
Set-Service docker -StartupType Disabled
```

Reboot the operating system prior executing the subsequent commands (`Restart-Computer`)

PowerShell

```
Get-NetNat | Remove-NetNat
Set-Service docker -StartupType Automatic
Start-Service docker
```

See this setup guide for multiple applications using the same NAT to rebuild your NAT environment, if necessary.

References

Read more about NAT networks

Create a virtual network

Article • 04/26/2022

Your virtual machines will need a virtual network to share a network with your computer. Creating a virtual network is optional -- if your virtual machine doesn't need to be connected to the internet or a network, skip ahead to creating a Windows Virtual Machine.

Connect virtual machines to the internet

Hyper-V has three types of virtual switches -- external, internal, and private. Create an external switch to share your computer's network with the virtual machines running on it.

This exercise walks through creating an external virtual switch. Once completed, your Hyper-V host will have a virtual switch that can connect virtual machines to the internet through your computer's network connection.

Create a Virtual Switch with Hyper-V Manager

1. Open Hyper-V Manager. A quick way to do this is by hitting the Windows button or key then type "Hyper-V Manager".
   If search doesn't find Hyper-V Manager, Hyper-V or the Hyper-V management tools are not enabled. See the instructions to enable Hyper-V.

2. Select the server in the left pane, or click "Connect to Server..." in the right pane.

3. In Hyper-V Manager, select **Virtual Switch Manager...** from the 'Actions' menu on the right.

4. Under the 'Virtual Switches' section, select **New virtual network switch**.

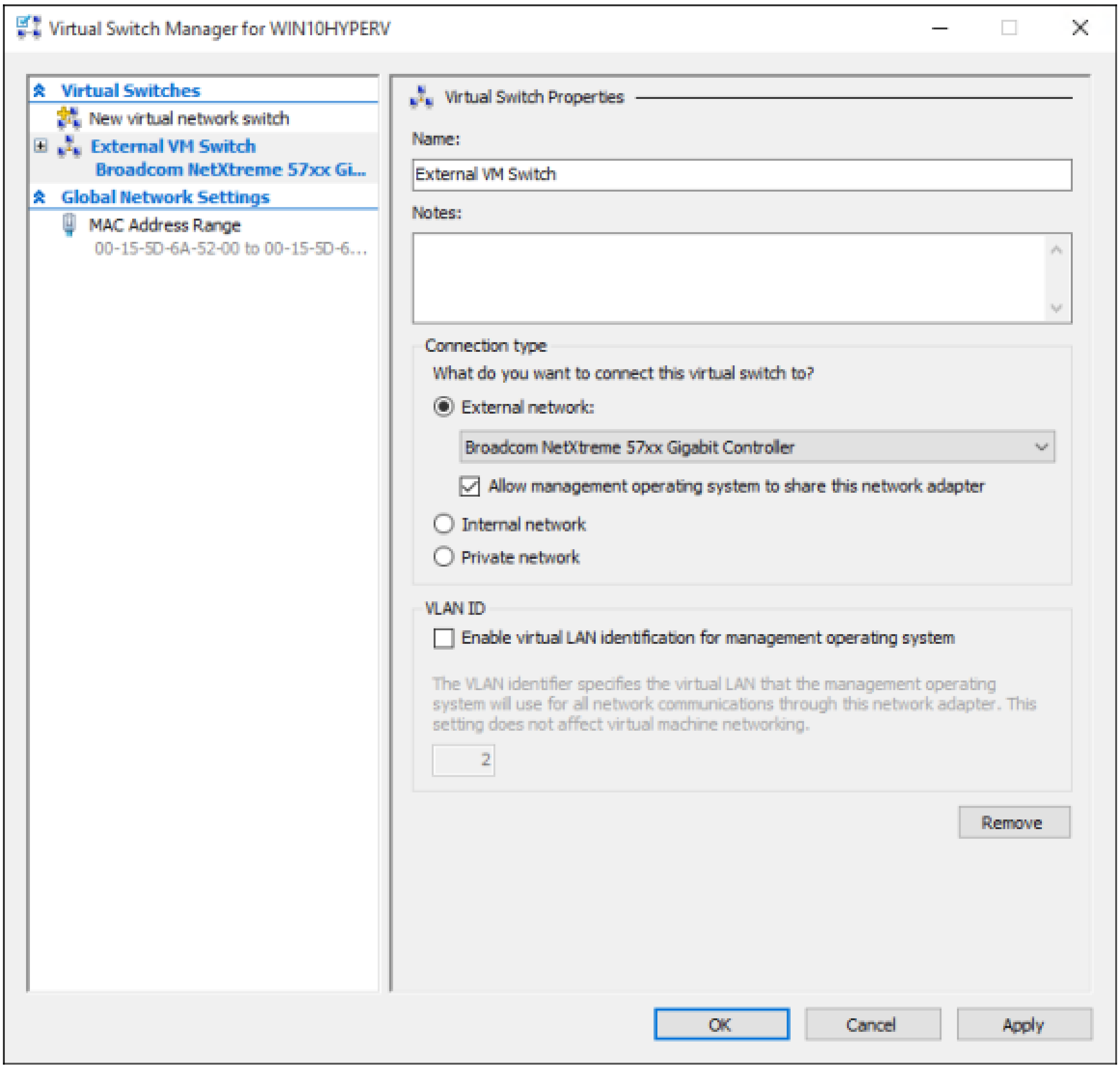
5. Under 'What type of virtual switch do you want to create?', select **External**.

6. Select the **Create Virtual Switch** button.

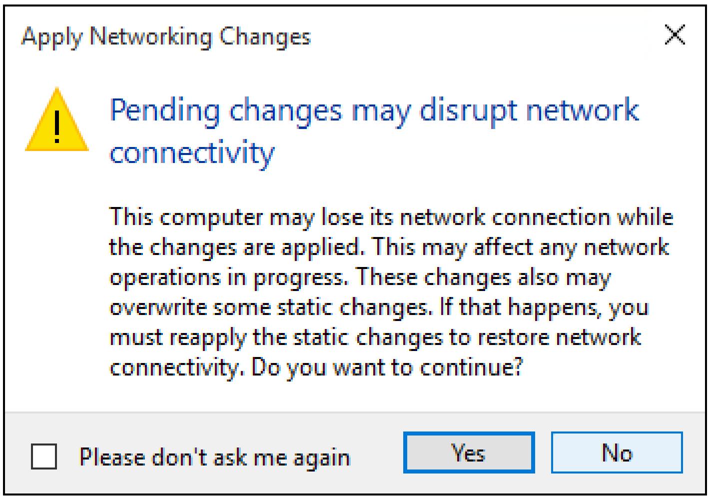
7. Under 'Virtual Switch Properties', give the new switch a name such as **External VM Switch**.

8. Under 'Connection Type', ensure that **External Network** has been selected.

9. Select the physical network card to be paired with the new virtual switch. This is the network card that is physically connected to the network.



10. Select **Apply** to create the virtual switch. At this point you will most likely see the following message. Click **Yes** to continue.



11. Select **OK** to close the Virtual Switch Manager Window.

Create a Virtual Switch with PowerShell

The following steps can be used to create a virtual switch with an external connection using PowerShell.

1. Use **Get-NetAdapter** to return a list of network adapters connected to the Windows 10 system.

PowerShell

```
PS C:\> Get-NetAdapter

Name                          InterfaceDescription
ifIndex Status       MacAddress             LinkSpeed
----                          --------------------                    -----
-- ------       ----------             ---------
Ethernet 2                    Broadcom NetXtreme 57xx Gigabit Cont...
5 Up          BC-30-5B-A8-C1-7F          1 Gbps
Ethernet                      Intel(R) PRO/100 M Desktop Adapter
3 Up          00-0E-0C-A8-DC-31         10 Mbps
```

2. Select the network adapter to use with the Hyper-V switch and place an instance in a variable named **$net**.

PowerShell

```
$net = Get-NetAdapter -Name 'Ethernet'
```

3. Execute the following command to create the new Hyper-V virtual switch.

PowerShell

```
New-VMSwitch -Name "External VM Switch" -AllowManagementOS $True -
NetAdapterName $net.Name
```

Virtual networking on a laptop

NAT networking

Network Address Translation (NAT) gives a virtual machine access to your computer's network by combining the host computer's IP address with a port through an internal Hyper-V Virtual Switch.

This has a few useful properties:

1. NAT Conserves IP addresses by mapping an external IP address and port to a much larger set of internal IP addresses.
2. NAT allows multiple virtual machines to host applications that require identical (internal) communication ports by mapping these to unique external ports.
3. NAT uses an internal switch -- creating an internal switch doesn't cause you to use network connection and tends to interfere less with a computer's networking.

To set up a NAT network and connect it to a virtual machine, follow the NAT networking user guide.

The two switch approach

If you’re running Windows 10 Hyper-V on a laptop and frequently switch between wireless networking and a wired network, you may want to create a virtual switch for both the ethernet and wireless network cards. Depending on how the laptop connects to the network, you can change your virtual machines between these switches. Virtual machines do not switch between wired and wireless automatically.

> ⓘ **Important**
>
> The two switch approach does not support External vSwitch over wireless card and should be used for testing purposes only.

Next Step - Create a Virtual Machine

Create a Windows Virtual Machine

Make your own integration services

Article • 01/17/2023

Starting in Windows 10 Anniversary Update, anyone can make applications that communicate between the Hyper-V host and its virtual machines using Hyper-V sockets -- a Windows Socket with a new address family and specialized endpoint for targeting virtual machines. All communication over Hyper-V sockets runs without using networking and all data stays on the same physical memory. Applications using Hyper-V sockets are similar to Hyper-V's integration services.

This document walks through creating a simple program built on Hyper-V sockets.

Supported Host OS

- Windows 10 and later
- Windows Server 2016 and later

Supported Guest OS

- Windows 10 and later
- Windows Server 2016 and later
- Linux guests with Linux Integration Services (see Supported Linux and FreeBSD virtual machines for Hyper-V on Windows)

> **Note:** A supported Linux guest must have kernel support for:
>
> Bash
>
> ```
> CONFIG_VSOCKET=y
> CONFIG_HYPERV_VSOCKETS=y
> ```

Capabilities and Limitations

- Supports kernel mode or user mode actions
- Data stream only
- No block memory (not the best for backup/video)

Getting started

Requirements:

- C/C++ compiler. If you don't have one, checkout Visual Studio Community
- Windows 10 SDK -- pre-installed in Visual Studio 2015 with Update 3 and later.
- A computer running one of the host operating systems above with at least one vitual machine. -- this is for testing your application.

> **Note:** The API for Hyper-V sockets became publicly available in Windows 10 Anniversary Update. Applications that use HVSocket will run on any Windows 10 host and guest but can only be developed with a Windows SDK later than build 14290.

Register a new application

In order to use Hyper-V sockets, the application must be registered with the Hyper-V Host's registry.

By registering the service in the registry, you get:

- WMI management for enable, disable, and listing available services
- Permission to communicate with virtual machines directly

The following PowerShell will register a new application named "HV Socket Demo". This must be run as administrator. Manual instructions below.

PowerShell

```powershell
$friendlyName = "HV Socket Demo"

# Create a new random GUID.  Add it to the services list
$service = New-Item -Path "HKLM:\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Virtualization\GuestCommunicationServices" -Name ((New-
Guid).Guid)

# Set a friendly name
$service.SetValue("ElementName", $friendlyName)

# Copy GUID to clipboard for later use
$service.PSChildName | clip.exe
```

Registry location and information:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Virtualization\GuestCommunicationServices\
```

In this registry location, you'll see several GUIDs. Those are our in-box services.

Information in the registry per service:

- `Service GUID`
  - `ElementName (REG_SZ)` -- this is the service's friendly name

To register your own service, create a new registry key using your own GUID and friendly name.

The friendly name will be associated with your new application. It will appear in performance counters and other places where a GUID isn't appropriate.

The registry entry will look like this:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Virtualization\GuestCommunicationServices\
    999E53D4-3D5C-4C3E-8779-BED06EC056E1\
        ElementName    REG_SZ    VM Session Service
    YourGUID\
        ElementName    REG_SZ    Your Service Friendly Name
```

> **Note:** The Service GUID for a Linux guest uses the VSOCK protocol which addresses via a `svm_cid` and `svm_port` rather than a guids. To bridge this inconsistency with Windows the well-known GUID is used as the service template on the host which translates to a port in the guest. To customize your Service GUID simply change the first "00000000" to the port number desired. Ex: "00000ac9" is port 2761.
>
> ```C++
> // Hyper-V Socket Linux guest VSOCK template GUID
> struct __declspec(uuid("00000000-facb-11e6-bd58-64006a7986d3"))
> VSockTemplate{};
>
>  /*
>   * GUID example = __uuidof(VSockTemplate);
>   * example.Data1 = 2761; // 0x00000AC9
>   */
> ```

> **Tip:** To generate a GUID in PowerShell and copy it to the clipboard, run:
>
> ```PowerShell
> (New-Guid).Guid | clip.exe
> ```

Create a Hyper-V socket

In the most basic case, defining a socket requires an address family, connection type, and protocol.

Here is a simple socket definition

C

```c
// Windows
SOCKET WSAAPI socket(
  _In_ int af,
  _In_ int type,
  _In_ int protocol
);

// Linux guest
int socket(int domain, int type, int protocol);
```

For a Hyper-V socket:

- Address family - `AF_HYPERV` (Windows) or `AF_VSOCK` (Linux guest)
- type - `SOCK_STREAM`
- protocol - `HV_PROTOCOL_RAW` (Windows) or `0` (Linux guest)

Here is an example declaration/instantiation:

C

```c
// Windows
SOCKET sock = socket(AF_HYPERV, SOCK_STREAM, HV_PROTOCOL_RAW);

// Linux guest
int sock = socket(AF_VSOCK, SOCK_STREAM, 0);
```

Bind to a Hyper-V socket

Bind associates a socket with connection information.

The function definition is copied below for convinience, read more about bind here.

C

```
// Windows
int bind(
  _In_ SOCKET                s,
  _In_ const struct sockaddr *name,
  _In_ int                   namelen
);

// Linux guest
int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

In contrast to the socket address (sockaddr) for a standard Internet Protocol address family (`AF_INET`) which consists of the host machine's IP address and a port number on that host, the socket address for `AF_HYPERV` uses the virtual machine's ID and the application ID defined above to establish a connection. If binding from a Linux guest `AF_VSOCK` uses the `svm_cid` and the `svm_port`.

Since Hyper-V sockets do not depend on a networking stack, TCP/IP, DNS, etc. the socket endpoint needed a non-IP, not hostname, format that still unambiguously describes the connection.

Here is the definition for a Hyper-V socket's socket address:

```C
// Windows
struct SOCKADDR_HV
{
     ADDRESS_FAMILY Family;
     USHORT Reserved;
     GUID VmId;
     GUID ServiceId;
};

// Linux guest
// See include/uapi/linux/vm_sockets.h for more information.
struct sockaddr_vm {
    __kernel_sa_family_t svm_family;
    unsigned short svm_reserved1;
    unsigned int svm_port;
    unsigned int svm_cid;
    unsigned char svm_zero[sizeof(struct sockaddr) -
                   sizeof(sa_family_t) -
                   sizeof(unsigned short) -
                   sizeof(unsigned int) - sizeof(unsigned int)];
};
```

In lieu of an IP or hostname, AF_HYPERV endpoints rely heavily on two GUIDs:

- VM ID – this is the unique ID assigned per VM. A VM's ID can be found using the following PowerShell snippet.

PowerShell

```powershell
(Get-VM -Name $VMName).Id
```

- Service ID – GUID, described above, with which the application is registered in the Hyper-V host registry.

There is also a set of VMID wildcards available when a connection isn't to a specific virtual machine.

VMID Wildcards

Name	GUID	Description
HV_GUID_ZERO	00000000-0000-0000-0000-000000000000	Listeners should bind to this VmId to accept connection from all partitions.
HV_GUID_WILDCARD	00000000-0000-0000-0000-000000000000	Listeners should bind to this VmId to accept connection from all partitions.
HV_GUID_BROADCAST	FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF	
HV_GUID_CHILDREN	90db8b89-0d35-4f79-8ce9-49ea0ac8b7cd	Wildcard address for children. Listeners should bind to this VmId to accept connection from its children.
HV_GUID_LOOPBACK	e0e16197-dd56-4a10-9195-5ee7a155a838	Loopback address. Using this VmId connects to the same partition as the connector.
HV_GUID_PARENT	a42e7cda-d03f-480c-9cc2-a4de20abb878	Parent address. Using this VmId connects to the parent partition of the connector.*

* `HV_GUID_PARENT` The parent of a virtual machine is its host. The parent of a container is the container's host. Connecting from a container running in a virtual machine will connect to the VM hosting the container. Listening on this VmId accepts connection from: (Inside containers): Container host. (Inside VM: Container host/ no container): VM host. (Not inside VM: Container host/ no container): Not supported.

Supported socket commands

Socket() Bind() Connect() Send() Listen() Accept()

HvSocket Socket Options

Name	Type	Description
HVSOCKET_CONNECTED_SUSPEND	ULONG	When this socket option is set to a non-zero value sockets do not disconnect when the virtual machine is paused.

Useful links

Complete WinSock API

Hyper-V Integration Services reference

Move from Hyper-V WMI v1 to WMI v2

Article • 04/26/2022

Windows Management Instrumentation (WMI) is the management interface underlying Hyper-V Manager and Hyper-V's PowerShell cmdlets. While most people use our PowerShell cmdlets or Hyper-V manager, sometimes developers needed WMI directly.

There have been two Hyper-V WMI namespaces (or versions of the Hyper-V WMI API).

- The WMI v1 namespace (root\virtualization) which was introduced in Windows Server 2008 and last available in Windows Server 2012
- The WMI v2 namespace (root\virtualization\v2) which was introduced in Windows Server 2012

This document contains references to resources for converting code that talks to our old WMI namespace to the new one. Initially, this article will serve as a repository for API information and sample code / scripts that can be used to help port any programs or scripts that use Hyper-V WMI APIs from the v1 namespace to the v2 namespace.

MSDN Samples

Hyper-V virtual machine migration sample
Hyper-V virtual Fiber Channel sample
Hyper-V planned virtual machines sample
Hyper-V application health monitoring sample
Virtual hard disk management sample
Hyper-V replication sample
Hyper-V metrics sample
Hyper-V dynamic memory sample
Hyper-V Extensible Switch extension filter driver
Hyper-V networking sample
Hyper-V resource pool management sample
Hyper-V recovery snapshot sample

Samples From Blogs

Adding a Network Adapter To A VM Using The Hyper-V WMI V2 Namespace
Connecting a VM Network Adapter To A Switch Using The Hyper-V WMI V2 Namespace
Changing The MAC Address Of NIC Using The Hyper-V WMI V2 Namespace

Removing a Network Adapter To A VM Using The Hyper-V WMI V2 Namespace
Attaching a VHD To A VM Using The Hyper-V WMI V2 Namespace
Removing a VHD From A VM Using The Hyper-V WMI V2 Namespace
Creating a VM using the Hyper-V WMI V2 Namespace

Windows 10 Hyper-V System Requirements

Article • 04/26/2022

Hyper-V is available in 64-bit version of Windows 10 Pro, Enterprise, and Education. Hyper-V requires Second Level Address Translation (SLAT) -- present in the current generation of 64-bit processors by Intel and AMD.

You can run 3 or 4 basic virtual machines on a host that has 4GB of RAM, though you'll need more resources for more virtual machines. On the other end of the spectrum, you can also create large virtual machines with 32 processors and 512GB RAM, depending on your physical hardware.

Operating System Requirements

The Hyper-V role can be enabled on these versions of Windows 10:

- Windows 10 Enterprise
- Windows 10 Pro
- Windows 10 Education

The Hyper-V role **cannot** be installed on:

- Windows 10 Home
- Windows 10 Mobile
- Windows 10 Mobile Enterprise

> Windows 10 Home edition can be upgraded to Windows 10 Pro. To do so open up **Settings** > **Update and Security** > **Activation**. Here you can visit the store and purchase an upgrade.

Hardware Requirements

Although this document does not provide a complete list of Hyper-V compatible hardware, the following items are necessary:

- 64-bit Processor with Second Level Address Translation (SLAT).
- CPU support for VM Monitor Mode Extension (VT-x on Intel CPU's).
- Minimum of 4 GB memory. As virtual machines share memory with the Hyper-V host, you will need to provide enough memory to handle the expected virtual

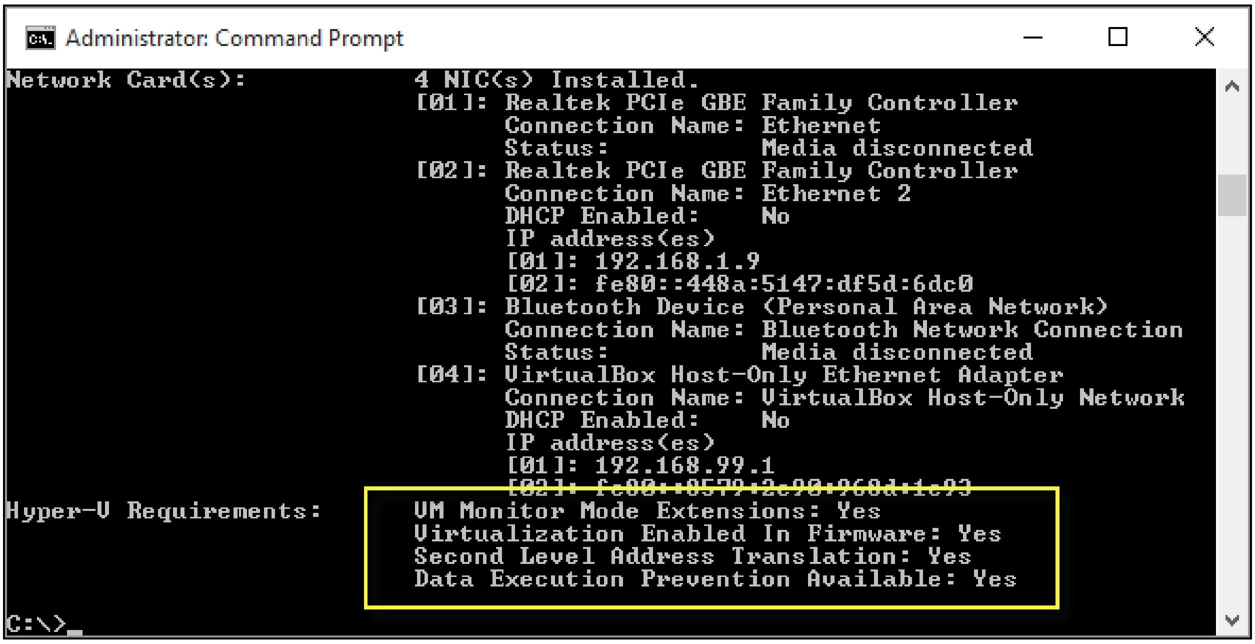
workload.

The following items will need to be enabled in the system BIOS:

- Virtualization Technology - may have a different label depending on motherboard manufacturer.
- Hardware Enforced Data Execution Prevention.

Verify Hardware Compatibility

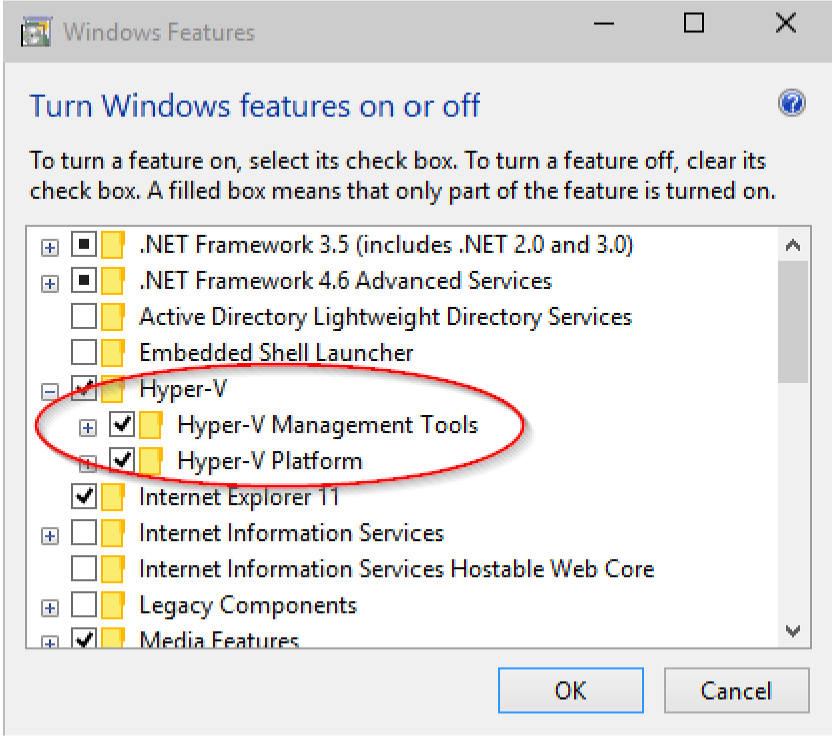
After checking the operating system and hardware requirements above, verify hardware compatibility in Windows by opening a PowerShell session or a command prompt (cmd.exe) window, typing **systeminfo**, and then checking the Hyper-V Requirements section. If all listed Hyper-V requirements have a value of **Yes**, your system can run the Hyper-V role. If any item returns **No**, check the requirements listed in this document and make adjustments where possible.



Final Check

If all OS, hardware and compatibility requirements are met then you will see **Hyper-V** in **Control Panel: Turn Windows features on or off** and it will have 2 options.

1. Hyper-V platform
2. Hyper-V Management Tools



ⓘ Note

If you see **Windows Hypervisor Platform** instead of **Hyper-V** in **Control Panel: Turn Windows features on or >off** your system may not be compatible for Hyper-V, then cross check above requirements. If you run **systeminfo** on an existing Hyper-V host, the Hyper-V Requirements section reads:

```
Hyper-V Requirements: A hypervisor has been detected. Features required
for Hyper-V will not be displayed.
```

Supported Windows guests

Article • 04/26/2022

This article lists the operating system combinations supported in Hyper-V on Windows. It also serves as an introduction to integration services and other factors in support.

Microsoft has tested these host/guest combinations. Issues with these combinations may receive attention from Product Support Services.

Microsoft provides support in the following manner:

- Issues found in Microsoft operating systems and in integration services are supported by Microsoft support.
- For issues found in other operating systems that have been certified by the operating system vendor to run on Hyper-V, support is provided by the vendor.
- For issues found in other operating systems, Microsoft submits the issue to the multi-vendor support community, TSANet.

In order to be supported, all operating systems (guest and host) must be up to date. Check Windows Update for critical updates.

Supported guest operating systems

Guest operating system	Maximum number of virtual processors	Notes
Windows 10	32	Enhanced Session Mode does not work on Windows 10 Home edition
Windows 8.1	32	
Windows 8	32	
Windows 7 with Service Pack 1 (SP 1)	4	Ultimate, Enterprise, and Professional editions (32-bit and 64-bit).
Windows 7	4	Ultimate, Enterprise, and Professional editions (32-bit and 64-bit).
Windows Vista with Service Pack 2 (SP2)	2	Business, Enterprise, and Ultimate, including N and KN editions.
Windows Server Semi-Annual Channel	64	

Guest operating system	Maximum number of virtual processors	Notes
Windows Server 2019	64	
Windows Server 2016	64	
Windows Server 2012 R2	64	
Windows Server 2012	64	
Windows Server 2008 R2 with Service Pack 1 (SP 1)	64	Datacenter, Enterprise, Standard and Web editions.
Windows Server 2008 with Service Pack 2 (SP 2)	4	Datacenter, Enterprise, Standard and Web editions (32-bit and 64-bit).
Windows Home Server 2011	4	
Windows Small Business Server 2011	Essentials edition - 2, Standard edition - 4	

> Windows 10 can run as a guest operating system on Windows 8.1 and Windows Server 2012 R2 Hyper-V hosts.

Supported Linux and FreeBSD

Guest operating system
CentOS and Red Hat Enterprise Linux
Debian virtual machines on Hyper-V
SUSE
Oracle Linux
Ubuntu
FreeBSD

For more information, including support information on past versions of Hyper-V, see Linux and FreeBSD Virtual Machines on Hyper-V.

Hyper-V

Reference

This reference provides cmdlet descriptions and syntax for all Hyper-V-specific cmdlets. It lists the cmdlets in alphabetical order based on the verb at the beginning of the cmdlet.

Hyper-V

Add-VMDvdDrive	Adds a DVD drive to a virtual machine.
Add-VMFibreChannelHba	Adds a virtual Fibre Channel host bus adapter to a virtual machine.
Add-VMGpuPartitionAdapter	Adds a GPU partition adapter to a virtual machine.
Add-VMGroupMember	Adds group members to a virtual machine group.
Add-VMHardDiskDrive	Adds a hard disk drive to a virtual machine.
Add-VMMigrationNetwork	Adds a network for virtual machine migration on one or more virtual machine hosts.
Add-VMNetworkAdapter	Adds a virtual network adapter to a virtual machine.
Add-VMNetworkAdapterAcl	Creates an ACL to apply to the traffic through a virtual machine network adapter.
Add-VMNetworkAdapterExtendedAcl	Creates an extended ACL for a virtual network adapter.
Add-VmNetworkAdapterRoutingDomainMapping	Adds a routing domain and virtual subnets to a virtual network adapter.
Add-VMRemoteFx3dVideoAdapter	Adds a RemoteFX video adapter in a virtual machine.
Add-VMScsiController	Adds a SCSI controller in a virtual machine.
Add-VMStoragePath	Adds a path to a storage resource pool.
Add-VMSwitch	Adds a virtual switch to an Ethernet resource pool.
Add-VMSwitchExtensionPortFeature	Adds a feature to a virtual network adapter.
Add-VMSwitchExtensionSwitchFeature	Adds a feature to a virtual switch.

Add-VMSwitchTeamMember	Adds members to a virtual switch team.
Checkpoint-VM	Creates a checkpoint of a virtual machine.
Compare-VM	Compares a virtual machine and a virtual machine host for compatibility, returning a compatibility report.
Complete-VMFailover	Completes a virtual machine's failover process on the Replica server. Removes all recovery points on a failed over virtual machine.
Connect-VMNetworkAdapter	Connects a virtual network adapter to a virtual switch.
Connect-VMSan	Associates a host bus adapter with a virtual storage area network (SAN).
Convert-VHD	Converts the format, version type, and block size of a virtual hard disk file.
Copy-VMFile	Copies a file to a virtual machine.
Debug-VM	Debugs a virtual machine.
Disable-VMConsoleSupport	Disables keyboard, video, and mouse for a generation 2 virtual machine.
Disable-VMEventing	Disables virtual machine eventing.
Disable-VMIntegrationService	Disables an integration service on a virtual machine.
Disable-VMMigration	Disables migration on one or more virtual machine hosts.
Disable-VMRemoteFXPhysicalVideoAdapter	Disables one or more RemoteFX physical video adapters from use with RemoteFX-enabled virtual machines.
Disable-VMResourceMetering	Disables collection of resource utilization data for a virtual machine or resource pool.
Disable-VMSwitchExtension	Disables one or more extensions on one or more virtual switches.
Disable-VMTPM	Disables TPM functionality on a virtual machine.
Disconnect-VMNetworkAdapter	Disconnects a virtual network adapter from a virtual switch or Ethernet resource pool.

Disconnect-VMSan	Removes a host bus adapter from a virtual storage area network (SAN).
Dismount-VHD	Dismounts a virtual hard disk.
Enable-VMConsoleSupport	Enables keyboard, video, and mouse for virtual machines.
Enable-VMEventing	Enables virtual machine eventing.
Enable-VMIntegrationService	Enables an integration service on a virtual machine.
Enable-VMMigration	Enables migration on one or more virtual machine hosts.
Enable-VMRemoteFXPhysicalVideoAdapter	Enables one or more RemoteFX physical video adapters for use with RemoteFX-enabled virtual machines.
Enable-VMReplication	Enables replication of a virtual machine.
Enable-VMResourceMetering	Collects resource utilization data for a virtual machine or resource pool.
Enable-VMSwitchExtension	Enables one or more extensions on one or more switches.
Enable-VMTPM	Enables TPM functionality on a virtual machine.
Export-VM	Exports a virtual machine to disk.
Export-VMSnapshot	Exports a virtual machine checkpoint to disk.
Get-VHD	Gets the virtual hard disk object associated with a virtual hard disk.
Get-VHDSet	Gets information about a VHD set.
Get-VHDSnapshot	Gets information about a checkpoint in a VHD set.
Get-VM	Gets the virtual machines from one or more Hyper-V hosts.
Get-VMBios	Gets the BIOS of a virtual machine or snapshot.
Get-VMComPort	Gets the COM ports of a virtual machine or snapshot.
Get-VMConnectAccess	Gets entries showing users and the virtual machines to which they can connect on one or more Hyper-V hosts.

Get-VMDvdDrive	Gets the DVD drives attached to a virtual machine or snapshot.
Get-VMFibreChannelHba	Gets the Fibre Channel host bus adapters associated with one or more virtual machines.
Get-VMFirmware	Gets the firmware configuration of a virtual machine.
Get-VMFloppyDiskDrive	Gets the floppy disk drives of a virtual machine or snapshot.
Get-VMGpuPartitionAdapter	Gets the information of assigned GPU partitions to a virtual machine.
Get-VMGroup	Gets virtual machine groups.
Get-VMHardDiskDrive	Gets the virtual hard disk drives attached to one or more virtual machines.
Get-VMHost	Gets a Hyper-V host.
Get-VMHostCluster	Gets virtual machine host clusters.
Get-VMHostNumaNode	Gets the NUMA topology of a virtual machine host.
Get-VMHostNumaNodeStatus	Gets the status of the virtual machines on the non-uniform memory access (NUMA) nodes of a virtual machine host or hosts.
Get-VMHostPartitionableGpu	Gets the host machine's partitionable GPU.
Get-VMHostSupportedVersion	Returns a list of virtual machine configuration versions that are supported on a host.
Get-VMIdeController	Gets the IDE controllers of a virtual machine or snapshot.
Get-VMIntegrationService	Gets the integration services of a virtual machine or snapshot.
Get-VMKeyProtector	Retrieves a key protector for a virtual machine.
Get-VMMemory	Gets the memory of a virtual machine or snapshot.
Get-VMMigrationNetwork	Gets the networks added for migration to one or more virtual machine hosts.
Get-VMNetworkAdapter	Gets the virtual network adapters of a virtual machine, snapshot, management operating

	system, or of a virtual machine and management operating system.
Get-VMNetworkAdapterAcl	Gets the ACLs configured for a virtual machine network adapter.
Get-VMNetworkAdapterExtendedAcl	Gets extended ACLs configured for a virtual network adapter.
Get-VMNetworkAdapterFailoverConfiguration	Gets the IP address of a virtual network adapter configured to be used when a virtual machine fails over.
Get-VmNetworkAdapterIsolation	Gets isolation settings for a virtual network adapter.
Get-VMNetworkAdapterRoutingDomainMapping	Gets members of a routing domain.
Get-VMNetworkAdapterTeamMapping	
Get-VMNetworkAdapterVlan	Gets the virtual LAN settings configured on a virtual network adapter.
Get-VMProcessor	Gets the processor of a virtual machine or snapshot.
Get-VMRemoteFx3dVideoAdapter	Gets the RemoteFX video adapter of a virtual machine or snapshot.
Get-VMRemoteFXPhysicalVideoAdapter	Gets the RemoteFX physical graphics adapters on one or more Hyper-V hosts.
Get-VMReplication	Gets the replication settings for a virtual machine.
Get-VMReplicationAuthorizationEntry	Gets the authorization entries of a Replica server.
Get-VMReplicationServer	Gets the replication and authentication settings of a Replica server.
Get-VMResourcePool	Gets the resource pools on one or more virtual machine hosts.
Get-VMSan	Gets the available virtual machine storage area networks on a Hyper-V host or hosts.
Get-VMScsiController	Gets the SCSI controllers of a virtual machine or snapshot.
Get-VMSecurity	Gets security information about a virtual machine.
Get-VMSnapshot	Gets the checkpoints associated with a virtual

	machine or checkpoint.
Get-VMStoragePath	Gets the storage paths in a storage resource pool.
Get-VMSwitch	Gets virtual switches from one or more virtual Hyper-V hosts.
Get-VMSwitchExtension	Gets the extensions on one or more virtual switches.
Get-VMSwitchExtensionPortData	Retrieves the status of a virtual switch extension feature applied to a virtual network adapter.
Get-VMSwitchExtensionPortFeature	Gets the features configured on a virtual network adapter.
Get-VMSwitchExtensionSwitchData	Gets the status of a virtual switch extension feature applied on a virtual switch.
Get-VMSwitchExtensionSwitchFeature	Gets the features configured on a virtual switch.
Get-VMSwitchTeam	Gets virtual switch teams from Hyper-V hosts.
Get-VMSystemSwitchExtension	Gets the switch extensions installed on a virtual machine host.
Get-VMSystemSwitchExtensionPortFeature	Gets the port-level features supported by virtual switch extensions on one or more Hyper-V hosts.
Get-VMSystemSwitchExtensionSwitchFeature	Gets the switch-level features on one or more Hyper-V hosts.
Get-VMVideo	Gets video settings for virtual machines.
Grant-VMConnectAccess	Grants a user or users access to connect to a virtual machine or machines.
Import-VM	Imports a virtual machine from a file.
Import-VMInitialReplication	Imports initial replication files for a Replica virtual machine to complete the initial replication when using external media as the source.
Measure-VM	Reports resource utilization data for one or more virtual machines.
Measure-VMReplication	Gets replication statistics and information associated with a virtual machine.
Measure-VMResourcePool	Reports resource utilization data for one or more resource pools.
Merge-VHD	Merges virtual hard disks.

Mount-VHD	Mounts one or more virtual hard disks.
Move-VM	Moves a virtual machine to a new Hyper-V host.
Move-VMStorage	Moves the storage of a virtual machine.
New-VFD	Creates a virtual floppy disk.
New-VHD	Creates one or more new virtual hard disks.
New-VM	Creates a new virtual machine.
New-VMGroup	Creates a virtual machine group. With Hyper-V, there are two types of VMGroups: a VMCollectionType and a ManagementCollectionType. A VMCollectionType VMGroup contains VMs while the ManagementCollectionType VMGroup contains VMCollectionType VMGroups. For example, you could have two VMCollectionType VMGroups VMG1 (containing VMs VM1 and VM2) and a second VMG2 (containing VMs VM3 and VM4). You could then create a ManagementCollectionType VMGroup VM-All containing the two VMCollectionType VMGroups. You use the Add-VMGroupMember cmdlet to add VMs to VMCollectionType VMGroups and to add VMCollectionType groups to ManagementCollectionType VMGroups.
New-VMReplicationAuthorizationEntry	Creates a new authorization entry that allows one or more primary servers to replicate data to a specified Replica server.
New-VMResourcePool	Creates a resource pool.
New-VMSan	Creates a new virtual storage area network (SAN) on a Hyper-V host.
New-VMSwitch	Creates a new virtual switch on one or more virtual machine hosts.
Optimize-VHD	Optimizes the allocation of space used by virtual hard disk files, except for fixed virtual hard disks.
Optimize-VHDSet	Optimizes VHD set files.
Remove-VHDSnapshot	Removes a checkpoint from a VHD set file.
Remove-VM	Deletes a virtual machine.

Remove-VMDvdDrive	Deletes a DVD drive from a virtual machine.
Remove-VMFibreChannelHba	Removes a Fibre Channel host bus adapter from a virtual machine.
Remove-VMGpuPartitionAdapter	Removes an assigned GPU partition from a virtual machine.
Remove-VMGroup	Removes a virtual machine group.
Remove-VMGroupMember	Removes members from a virtual machine group.
Remove-VMHardDiskDrive	Deletes a hard disk drive from a virtual machine.
Remove-VMMigrationNetwork	Removes a network from use with migration.
Remove-VMNetworkAdapter	Removes one or more virtual network adapters from a virtual machine.
Remove-VMNetworkAdapterAcl	Removes an ACL applied to the traffic through a virtual network adapter.
Remove-VMNetworkAdapterExtendedAcl	Removes an extended ACL for a virtual network adapter.
Remove-VMNetworkAdapterRoutingDomainMapping	Removes a routing domain from a virtual network adapter.
Remove-VMNetworkAdapterTeamMapping	
Remove-VMRemoteFx3dVideoAdapter	Removes a RemoteFX 3D video adapter from a virtual machine.
Remove-VMReplication	Removes the replication relationship of a virtual machine.
Remove-VMReplicationAuthorizationEntry	Removes an authorization entry from a Replica server.
Remove-VMResourcePool	Deletes a resource pool from one or more virtual machine hosts.
Remove-VMSan	Removes a virtual storage area network (SAN) from a Hyper-V host.
Remove-VMSavedState	Deletes the saved state of a saved virtual machine.
Remove-VMScsiController	Removes a SCSI controller from a virtual machine.
Remove-VMSnapshot	Deletes a virtual machine checkpoint.
Remove-VMStoragePath	Removes a path from a storage resource pool.

Remove-VMSwitch	Deletes a virtual switch.
Remove-VMSwitchExtensionPortFeature	Removes a feature from a virtual network adapter.
Remove-VMSwitchExtensionSwitchFeature	Removes a feature from a virtual switch.
Remove-VMSwitchTeamMember	Removes a member from a virtual machine switch team.
Rename-VM	Renames a virtual machine.
Rename-VMGroup	Renames virtual machine groups.
Rename-VMNetworkAdapter	Renames a virtual network adapter on a virtual machine or on the management operating system.
Rename-VMResourcePool	Renames a resource pool on one or more Hyper-V hosts.
Rename-VMSan	Renames a virtual storage area network (SAN).
Rename-VMSnapshot	Renames a virtual machine checkpoint.
Rename-VMSwitch	Renames a virtual switch.
Repair-VM	Repairs one or more virtual machines.
Reset-VMReplicationStatistics	Resets the replication statistics of a virtual machine.
Reset-VMResourceMetering	Resets the resource utilization data collected by Hyper-V resource metering.
Resize-VHD	Resizes a virtual hard disk.
Restart-VM	Restarts a virtual machine.
Restore-VMSnapshot	Restores a virtual machine checkpoint.
Resume-VM	Resumes a suspended (paused) virtual machine.
Resume-VMReplication	Resumes a virtual machine replication that is in a state of Paused, Error, Resynchronization Required, or Suspended.
Revoke-VMConnectAccess	Revokes access for one or more users to connect to a one or more virtual machines.
Save-VM	Saves a virtual machine.
Set-VHD	Sets properties associated with a virtual hard disk.
Set-VM	Configures a virtual machine.

Set-VMBios	Configures the BIOS of a Generation 1 virtual machine.
Set-VMComPort	Configures the COM port of a virtual machine.
Set-VMDvdDrive	Configures a virtual DVD drive.
Set-VMFibreChannelHba	Configures a Fibre Channel host bus adapter on a virtual machine.
Set-VMFirmware	Sets the firmware configuration of a virtual machine.
Set-VMFloppyDiskDrive	Configures a virtual floppy disk drive.
Set-VMGpuPartitionAdapter	Assigns a partition of a GPU to a virtual machine.
Set-VMHardDiskDrive	Configures a virtual hard disk.
Set-VMHost	Configures a Hyper-V host.
Set-VMHostCluster	Configures a virtual machine host cluster.
Set-VMHostPartitionableGpu	Configures the host partitionable GPU to the number of partitions supported by the manufacturer.
Set-VMKeyProtector	Configures a key protector for a virtual machine.
Set-VMMemory	Configures the memory of a virtual machine.
Set-VMMigrationNetwork	Sets the subnet, subnet mask, and/or priority of a migration network.
Set-VMNetworkAdapter	Configures features of the virtual network adapter in a virtual machine or the management operating system.
Set-VMNetworkAdapterFailoverConfiguration	Configures the IP address of a virtual network adapter to be used when a virtual machine fails over.
Set-VmNetworkAdapterIsolation	Modifies isolation settings for a virtual network adapter.
Set-VmNetworkAdapterRoutingDomainMapping	Sets virtual subnets on a routing domain.
Set-VMNetworkAdapterTeamMapping	
Set-VMNetworkAdapterVlan	Configures the virtual LAN settings for the traffic through a virtual network adapter.

Set-VMProcessor	Configures settings for the virtual processors of a virtual machine. Settings are applied uniformly to all virtual processors belonging to the virtual machine.
Set-VMRemoteFx3dVideoAdapter	Configures the RemoteFX 3D video adapter of a virtual machine.
Set-VMReplication	Modifies the replication settings of a virtual machine.
Set-VMReplicationAuthorizationEntry	Modifies an authorization entry on a Replica server.
Set-VMReplicationServer	Configures a host as a Replica server.
Set-VMResourcePool	Sets the parent resource pool for a selected resource pool.
Set-VMSan	Configures a virtual storage area network (SAN) on one or more Hyper-V hosts.
Set-VMSecurity	Configures security settings for a virtual machine.
Set-VMSecurityPolicy	Configures the security policy for a virtual machine.
Set-VMSwitch	Configures a virtual switch.
Set-VMSwitchExtensionPortFeature	Configures a feature on a virtual network adapter.
Set-VMSwitchExtensionSwitchFeature	Configures a feature on a virtual switch.
Set-VMSwitchTeam	Configures a virtual switch team.
Set-VMVideo	Configures video settings for virtual machines.
Start-VM	Starts a virtual machine.
Start-VMFailover	Starts failover on a virtual machine.
Start-VMInitialReplication	Starts replication of a virtual machine.
Start-VMTrace	Starts tracing to a file.
Stop-VM	Shuts down, turns off, or saves a virtual machine.
Stop-VMFailover	Stops failover of a virtual machine.
Stop-VMInitialReplication	Stops an ongoing initial replication.
Stop-VMReplication	Cancels an ongoing virtual machine resynchronization.

Stop-VMTrace	Stops tracing to file.
Suspend-VM	Suspends, or pauses, a virtual machine.
Suspend-VMReplication	Suspends replication of a virtual machine.
Test-VHD	Tests a virtual hard disk for any problems that would make it unusable.
Test-VMNetworkAdapter	Tests connectivity between virtual machines.
Test-VMReplicationConnection	Tests the connection between a primary server and a Replica server.
Update-VMVersion	Updates the version of virtual machines.

Hyper-V Integration Services

Article • 04/26/2022

Integration services (often called integration components), are services that allow the virtual machine to communicate with the Hyper-V host. Many of these services are conveniences while others can be quite important to the virtual machine's ability to function correctly.

This article is a reference for each integration service available in Windows. It will also act as a starting point for any information related to specific integration services or their history.

User Guides:

- Managing integration services

Quick Reference

Name	Windows Service Name	Linux Daemon Name	Description	Impact on VM when disabled
Hyper-V Heartbeat Service	vmicheartbeat	hv_utils	Reports that the virtual machine is running correctly.	Varies
Hyper-V Guest Shutdown Service	vmicshutdown	hv_utils	Allows the host to trigger virtual machines shutdown.	**High**
Hyper-V Time Synchronization Service	vmictimesync	hv_utils	Synchronizes the virtual machine's clock with the host computer's clock.	**High**
Hyper-V Data Exchange Service (KVP)	vmickvpexchange	hv_kvp_daemon	Provides a way to exchange basic metadata between the virtual machine and the host.	Medium
Hyper-V Volume Shadow Copy Requestor	vmicvss	hv_vss_daemon	Allows Volume Shadow Copy Service to back up the virtual machine with out shutting it down.	Varies

Name	Windows Service Name	Linux Daemon Name	Description	Impact on VM when disabled
Hyper-V Guest Service Interface	vmicguestinterface	hv_fcopy_daemon	Provides an interface for the Hyper-V host to copy files to or from the virtual machine.	Low
Hyper-V PowerShell Direct Service	vmicvmsession	not available	Provides a way to manage virtual machine with PowerShell without a network connection.	Low

Hyper-V Heartbeat Service

Windows Service Name: vmicheartbeat
Linux Daemon Name: hv_utils
Description: Tells the Hyper-V host that the virtual machine has an operating system installed and that it booted correctly.
Added In: Windows Server 2012, Windows 8
Impact: When disabled, the virtual machine can't report that the operating system inside of the virtual machine is operating correctly. This may impact some kinds of monitoring and host-side diagnostics.

The heartbeat service makes it possible to answer basic questions like "did the virtual machine boot?".

When Hyper-V reports that a virtual machine state is "running" (see the example below), it means Hyper-V set aside resources for a virtual machine; it does not mean that there is an operating system installed or functioning. This is where heartbeat becomes useful. The heartbeat service tells Hyper-V that the operating system inside the virtual machine has booted.

Check heartbeat with PowerShell

Run Get-VM as Administrator to see a virtual machine's heartbeat:

PowerShell

```
Get-VM -VMName $VMName | select Name, State, Status
```

Your output should look something like this:

```
Name    State    Status
----    -----    ------
DemoVM  Running  Operating normally
```

The `Status` field is determined by the heartbeat service.

Hyper-V Guest Shutdown Service

Windows Service Name: vmicshutdown
Linux Daemon Name: hv_utils
Description: Allows the Hyper-V host to request that the virtual machine shutdown. The host can always force the virtual machine to turn off, but that is like flipping the power switch as opposed to selecting shutdown.
Added In: Windows Server 2012, Windows 8
Impact: High Impact When disabled, the host can't trigger a friendly shutdown inside the virtual machine. All shutdowns will be a hard power-off, which could cause data loss or data corruption.

Hyper-V Time Synchronization Service

Windows Service Name: vmictimesync
Linux Daemon Name: hv_utils
Description: Synchronizes the virtual machine's system clock with the system clock of the physical computer.
Added In: Windows Server 2012, Windows 8
Impact: High Impact When disabled, the virtual machine's clock will drift erratically.

Hyper-V Data Exchange Service (KVP)

Windows Service Name: vmickvpexchange
Linux Daemon Name: hv_kvp_daemon
Description: Provides a mechanism to exchange basic metadata between the virtual machine and the host.
Added In: Windows Server 2012, Windows 8
Impact: When disabled, virtual machines running Windows 8 or Windows Server 2012 or

earlier will not receive updates to Hyper-V integration services. Disabling data exchange may also impact some kinds of monitoring and host-side diagnostics.

The data exchange service (sometimes called KVP) shares small amounts of machine information between virtual machine and the Hyper-V host using key-value pairs (KVP) through the Windows registry. The same mechanism can also be used to share customized data between the virtual machine and the host.

Key-value pairs consist of a “key” and a “value”. Both the key and the value are strings, no other data types are supported. When a key-value pair is created or changed, it is visible to the guest and the host. The key-value pair information is transferred across the Hyper-V VMbus and does not require any kind of network connection between the guest and the Hyper-V host.

The data exchange service is a great tool for preserving information about the virtual machine -- for interactive data sharing or data transfer, use PowerShell Direct.

User Guides:

- Using key-value pairs to share information between the host and guest on Hyper-V.

Hyper-V Volume Shadow Copy Requestor

Windows Service Name: vmicvss
Linux Daemon Name: hv_vss_daemon
Description: Allows Volume Shadow Copy Service to back up applications and data on the virtual machine.
Added In: Windows Server 2012, Windows 8
Impact: When disabled, the virtual machine can not be backed up while running (using VSS).

The Volume Shadow Copy Requestor integration service is required for Volume Shadow Copy Service (VSS). The Volume Shadow Copy Service (VSS) captures and copies images for backup on running systems, particularly servers, without unduly degrading the performance and stability of the services they provide. This integration service makes that possible by coordinating the virtual machine's workloads with the host's backup process.

Read more about Volume Shadow Copy here.

Hyper-V Guest Service Interface

Windows Service Name: vmicguestinterface
Linux Daemon Name: hv_fcopy_daemon
Description: Provides an interface for the Hyper-V host to bidirectionally copy files to or from the virtual machine.
Added In: Windows Server 2012 R2, Windows 8.1
Impact: When disabled, the host can not copy files to and from the guest using `Copy-VMFile`. Read more about the Copy-VMFile cmdlet.

Notes:
Disabled by default. See PowerShell Direct using Copy-Item.

Hyper-V PowerShell Direct Service

Windows Service Name: vmicvmsession
Linux Daemon Name: n/a
Description: Provides a mechanism to manage virtual machine with PowerShell via VM session without a virtual network.
Added In: Windows Server TP3, Windows 10
Impact: Disabling this service prevents the host from being able to connect to the virtual machine with PowerShell Direct.

Notes:
The service name was originally was Hyper-V VM Session Service.
PowerShell Direct is under active development and only available on Windows 10/Windows Server Technical Preview 3 or later hosts/guests.

PowerShell Direct allows PowerShell management inside a virtual machine from the Hyper-V host regardless of any network configuration or remote management settings on either the Hyper-V host or the virtual machine. This makes it easier for Hyper-V Administrators to automate and script management and configuration tasks.

Read more about PowerShell Direct.

User Guides:

- Running script in a virtual machine
- Copying files to and from a virtual machine

Hyper-V Architecture

Article • 04/26/2022

Hyper-V is a hypervisor-based virtualization technology for certain x64 versions of Windows. The hypervisor is core to virtualization. It is the processor-specific virtualization platform that allows multiple isolated operating systems to share a single hardware platform.

Hyper-V supports isolation in terms of a partition. A partition is a logical unit of isolation, supported by the hypervisor, in which operating systems execute. The Microsoft hypervisor must have at least one parent, or root, partition, running Windows. The virtualization management stack runs in the parent partition and has direct access to hardware devices. The root partition then creates the child partitions which host the guest operating systems. A root partition creates child partitions using the hypercall application programming interface (API).

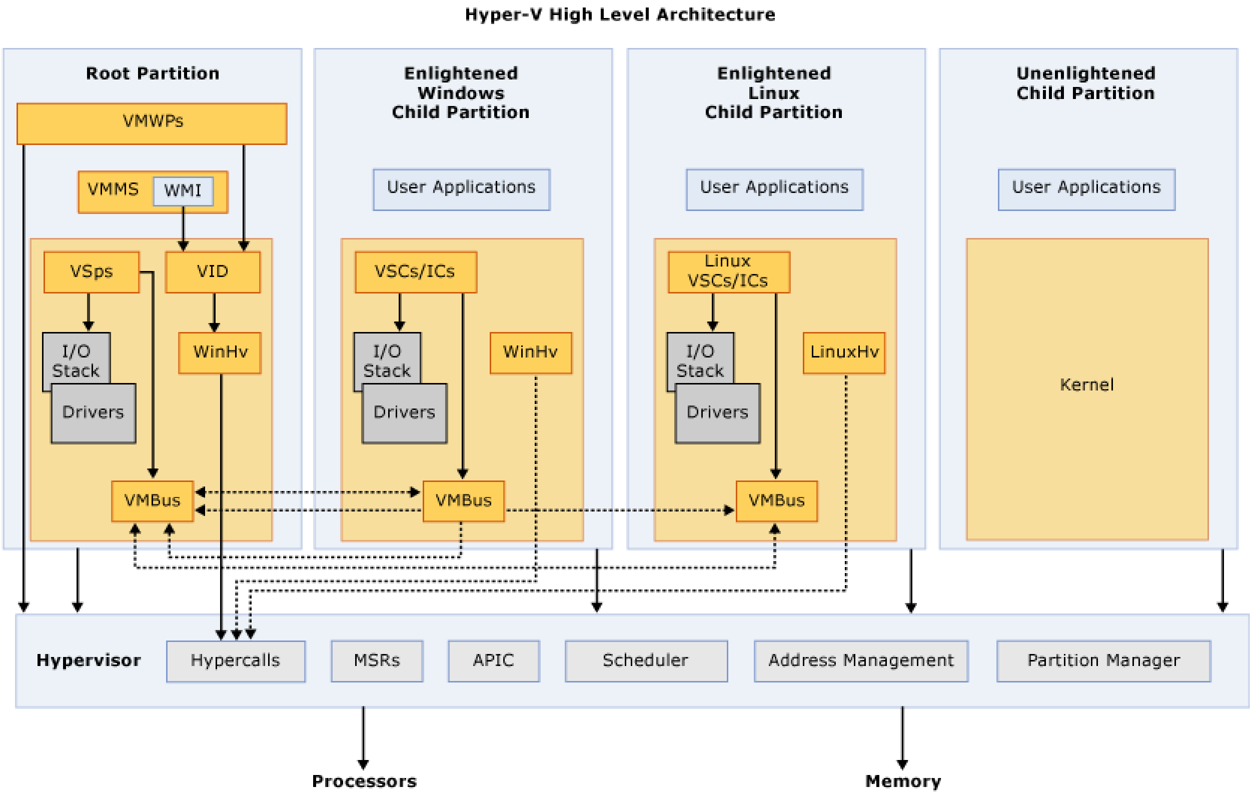
Partitions do not have access to the physical processor, nor do they handle the processor interrupts. Instead, they have a virtual view of the processor and run in a virtual memory address region that is private to each guest partition. The hypervisor handles the interrupts to the processor, and redirects them to the respective partition. Hyper-V can also hardware accelerate the address translation between various guest virtual address spaces by using an Input Output Memory Management Unit (IOMMU) which operates independent of the memory management hardware used by the CPU. An IOMMU is used to remap physical memory addresses to the addresses that are used by the child partitions.

Child partitions also do not have direct access to other hardware resources and are presented a virtual view of the resources, as virtual devices (VDevs). Requests to the virtual devices are redirected either via the VMBus or the hypervisor to the devices in the parent partition, which handles the requests. The VMBus is a logical inter-partition communication channel. The parent partition hosts Virtualization Service Providers (VSPs) which communicate over the VMBus to handle device access requests from child partitions. Child partitions host Virtualization Service Consumers (VSCs) which redirect device requests to VSPs in the parent partition via the VMBus. This entire process is transparent to the guest operating system.

Virtual Devices can also take advantage of a Windows Server Virtualization feature, named Enlightened I/O, for storage, networking, graphics, and input subsystems. Enlightened I/O is a specialized virtualization-aware implementation of high level communication protocols (such as SCSI) that utilize the VMBus directly, bypassing any device emulation layer. This makes the communication more efficient but requires an

enlightened guest that is hypervisor and VMBus aware. Hyper-V enlightened I/O and a hypervisor aware kernel is provided via installation of Hyper-V integration services. Integration components, which include virtual server client (VSC) drivers, are also available for other client operating systems. Hyper-V requires a processor that includes hardware assisted virtualization, such as is provided with Intel VT or AMD Virtualization (AMD-V) technology.

The following diagram provides a high-level overview of the architecture of a Hyper-V environment.



Glossary

- **APIC** – Advanced Programmable Interrupt Controller – A device which allows priority levels to be assigned to its interrupt outputs.
- **Child Partition** – Partition that hosts a guest operating system - All access to physical memory and devices by a child partition is provided via the Virtual Machine Bus (VMBus) or the hypervisor.
- **Hypercall** – Interface for communication with the hypervisor - The hypercall interface accommodates access to the optimizations provided by the hypervisor.
- **Hypervisor** – A layer of software that sits between the hardware and one or more operating systems. Its primary job is to provide isolated execution environments called partitions. The hypervisor controls and arbitrates access to the underlying hardware.

- **IC** – Integration component – Component that allows child partitions to communication with other partitions and the hypervisor.
- **I/O stack** – Input/output stack
- **MSR** – Memory Service Routine
- **Root Partition** – Sometimes called parent partition. Manages machine-level functions such as device drivers, power management, and device hot addition/removal. The root (or parent) partition is the only partition that has direct access to physical memory and devices.
- **VID** – Virtualization Infrastructure Driver – Provides partition management services, virtual processor management services, and memory management services for partitions.
- **VMBus** – Channel-based communication mechanism used for inter-partition communication and device enumeration on systems with multiple active virtualized partitions. The VMBus is installed with Hyper-V Integration Services.
- **VMMS** – Virtual Machine Management Service – Responsible for managing the state of all virtual machines in child partitions.
- **VMWP** – Virtual Machine Worker Process – A user mode component of the virtualization stack. The worker process provides virtual machine management services from the Windows Server 2008 instance in the parent partition to the guest operating systems in the child partitions. The Virtual Machine Management Service spawns a separate worker process for each running virtual machine.
- **VSC** – Virtualization Service Client – A synthetic device instance that resides in a child partition. VSCs utilize hardware resources that are provided by Virtualization Service Providers (VSPs) in the parent partition. They communicate with the corresponding VSPs in the parent partition over the VMBus to satisfy a child partitions device I/O requests.
- **VSP** – Virtualization Service Provider – Resides in the root partition and provide synthetic device support to child partitions over the Virtual Machine Bus (VMBus).
- **WinHv** – Windows Hypervisor Interface Library - WinHv is essentially a bridge between a partitioned operating system’s drivers and the hypervisor which allows drivers to call the hypervisor using standard Windows calling conventions
- **WMI** – The Virtual Machine Management Service exposes a set of Windows Management Instrumentation (WMI)-based APIs for managing and controlling virtual machines.

Hypervisor Top Level Functional Specification

Article • 05/20/2022

The Hyper-V Hypervisor Top-Level Functional Specification (TLFS) describes the hypervisor's guest-visible behavior to other operating system components. This specification is meant to be useful for guest operating system developers.

> This specification is provided under the Microsoft Open Specification Promise. Read the following for further details about the Microsoft Open Specification Promise.

Glossary

- **Partition** - Hyper-V supports isolation in terms of a partition. A partition is a logical unit of isolation, supported by the hypervisor, in which operating systems execute.
- **Root Partition** - The root partition (a.k.a the"parent" or"host") is a privileged management partition. The root partition manages machine-level functions such as device drivers, power management, and device addition/removal. The virtualization stack runs in the parent partition and has direct access to the hardware devices. The root partition then creates the child partitions which host the guest operating systems.
- **Child Partition** - The child partition (a.k.a. the"guest") hosts a guest operating system. All access to physical memory and devices by a child partition is provided via the Virtual Machine Bus (VMBus) or the hypervisor.
- **Hypercall** - Hypercalls are an interface for communication with the hypervisor.

Specification Style

The document assumes familiarity with the high-level hypervisor architecture.

This specification is informal; that is, the interfaces are not specified in a formal language. Nevertheless, it is a goal to be precise. It is also a goal to specify which

behaviors are architectural and which are implementation-specific. Callers should not rely on behaviors that fall into the latter category because they may change in future implementations.

Previous Versions

Release	Document
Windows Server 2016 (Revision C)	Hypervisor Top Level Functional Specification v5.0c.pdf
Windows Server 2012 R2 (Revision B)	Hypervisor Top Level Functional Specification v4.0b.pdf
Windows Server 2012	Hypervisor Top Level Functional Specification v3.0.pdf
Windows Server 2008 R2	Hypervisor Top Level Functional Specification v2.0.pdf

Requirements for Implementing the Microsoft Hypervisor Interface

The TLFS fully describes all aspects of the Microsoft-specific hypervisor architecture, which is declared to guest virtual machines as the "HV#1" interface. However, not all interfaces described in the TLFS are required to be implemented by third-party hypervisor wishing to declare conformance with the Microsoft HV#1 hypervisor specification. The document "Requirements for Implementing the Microsoft Hypervisor Interface" describes the minimal set of hypervisor interfaces which must be implemented by any hypervisor which claims compatibility with the Microsoft HV#1 interface.

Requirements for Implementing the Microsoft Hypervisor Interface.pdf

Data Types

Article • 05/26/2021

Reserved Values

This specification documents some fields as “reserved.” These fields may be given specific meaning in future versions of the hypervisor architecture. For maximum forward compatibility, clients of the hypervisor interface should follow the guidance provided within this document. In general, two forms of guidance are provided. Preserve value (documented as RsvdP in diagrams and ReservedP in code segments) – For maximum forward compatibility, clients should preserve the value within this field. This is typically done by reading the current value, modifying the values of the non-reserved fields, and writing the value back. Zero value (documented as RsvdZ in diagrams and ReservedZ in code segments) – For maximum forward compatibility, clients should zero the value within this field.

Reserved fields within read-only structures are simply documented as Rsvd in diagrams and simply as Reserved in code segments. For maximum forward compatibility, the values within these fields should be ignored. Clients should not assume these values will always be zero.

Simple Scalar Types

Hypervisor data types are built up from simple scalar types UINT8, UINT16, UINT32, UINT64 and UINT128. Each of these represents a simple unsigned integer scalar with the specified bit count. Several corresponding signed integer scalars are also defined: INT8, INT16, INT32, and INT64. The hypervisor uses neither floating point instructions nor floating point types.

Hypercall Status Code

Every hypercall returns a 16-bit status code of type HV_STATUS.

```C
typedef UINT16 HV_STATUS;
```

Memory Address Space Types

The hypervisor architecture defines three independent address spaces:

- **System physical addresses (SPAs)** define the physical address space of the underlying hardware as seen by the CPUs. There is only one system physical address space for the entire machine.
- **Guest physical addresses (GPAs)** define the guest's view of physical memory. GPAs can be mapped to underlying SPAs. There is one guest physical address space per partition.
- **Guest virtual addresses (GVAs)** are used within the guest when it enables address translation and provides a valid guest page table.

All three of these address spaces are up to 264 bytes in size. The following types are thus defined:

```C
typedef UINT64 HV_SPA;
typedef UINT64 HV_GPA;
typedef UINT64 HV_GVA;
```

Many hypervisor interfaces act on pages of memory rather than single bytes. The minimum page size is architecture-dependent. For x64, it is defined as 4K.

```C
#define X64_PAGE_SIZE 0x1000

#define HV_X64_MAX_PAGE_NUMBER (MAXUINT64/X64_PAGE_SIZE)
#define HV_PAGE_SIZE X64_PAGE_SIZE
#define HV_LARGE_PAGE_SIZE X64_LARGE_PAGE_SIZE
#define HV_PAGE_MASK (HV_PAGE_SIZE - 1)

typedef UINT64 HV_SPA_PAGE_NUMBER;
typedef UINT64 HV_GPA_PAGE_NUMBER;
typedef UINT64 HV_GVA_PAGE_NUMBER;
typedef UINT32 HV_SPA_PAGE_OFFSET
typedef HV_GPA_PAGE_NUMBER *PHV_GPA_PAGE_NUMBER;
```

To convert an `HV_SPA` to an `HV_SPA_PAGE_NUMBER`, simply divide by `HV_PAGE_SIZE`.

Structures, Enumerations and Bit Fields

Many data structures and constant values defined later in this specification are defined in terms of C-style enumerations and structures. The C language purposely avoids defining certain implementation details. However, this document assumes the following:

- All enumerations declared with the “enum” keyword define 32-bit signed integer values.
- All structures are padded in such a way that fields are aligned naturally (that is, an 8-byte field is aligned to an offset of 8 bytes and so on).
- All bit fields are packed from low-order to high-order bits with no padding.

Endianness

The hypervisor interface is designed to be endian-neutral (that is, it should be possible to port the hypervisor to a big-endian or little-endian system), but some of the data structures defined later in this specification assume little-endian layout. Such data structures will need to be amended if and when a big-endian port is attempted.

Pointer Naming Convention

The document uses a naming convention for pointer types. In particular, a “P” prepended to a defined type indicates a pointer to that type. A “PC” prepended to a defined type indicates a pointer to a constant value of that type.

Feature and Interface Discovery

Article • 07/08/2022

Guest software interacts with the hypervisor through a variety of mechanisms. Many of these mirror the traditional mechanisms used by software to interact with the underlying processor. As such, these mechanisms are architecture-specific. On the x64 architecture, the following mechanisms are used:

- CPUID instruction – Used for static feature and version information.
- MSRs (model-specific registers) – Used for status and control values.
- Memory-mapped registers – Used for status and control values.
- Processor interrupts – Used for asynchronous events, notifications and messages.

In addition to these architecture-specific interfaces, the hypervisor provides a simple procedural interface implemented with hypercalls.

Hypervisor Discovery

Before using any hypervisor interfaces, software should first determine whether it's running within a virtualized environment. On x64 platforms that conform to this specification, this is done by executing the CPUID instruction with an input (EAX) value of 1. Upon execution, code should check bit 31 of register ECX (the "hypervisor present bit"). If this bit is set, a hypervisor is present. In a non-virtualized environment, the bit will be clear.

```c
CPUID.01h.ECX:31 // if set, virtualization present
```

If the "hypervisor present bit" is set, additional CPUID leafs can be queried for more information about the conformant hypervisor and its capabilities. Two such leaves are guaranteed to be available: `0x40000000` and `0x40000001`. Subsequently-numbered leaves may also be available.

Standard Hypervisor CPUID Leaves

When the leaf at `0x40000000` is queried, the hypervisor will return information that provides the maximum hypervisor CPUID leaf number and a vendor ID signature.

Register	Information Provided

Register	Information Provided
EAX	The maximum input value for hypervisor CPUID information
EBX	Hypervisor Vendor ID Signature
ECX	Hypervisor Vendor ID Signature
EDX	Hypervisor Vendor ID Signature

If the leaf at `0x40000001` is queried, it will return a value representing a vendor-neutral hypervisor interface identification. This determines the semantics of the leaves from `0x4000002` through `0x400000FF`.

Register	Information Provided
EAX	Hypervisor Interface Signature.
EBX	Reserved
ECX	Reserved
EDX	Reserved

These two leaves allow the guest to query the hypervisor vendor ID and interface independently. The vendor ID is provided only for informational and diagnostic purposes. It is recommended that software only base compatibility decisions on the interface signature reported through leaf `0x40000001`.

Microsoft Hypervisor CPUID Leaves

On hypervisors conforming to the Microsoft hypervisor CPUID interface, the `0x40000000` and `0x40000001` leaf registers will have the following values.

Hypervisor CPUID Leaf Range - 0x40000000

EAX determines the maximum hypervisor CPUID leaf. EBX-EDX contain the hypervisor vendor ID signature. The vendor ID signature should be used only for reporting and diagnostic purposes.

Register	Information Provided
EAX	The maximum input value for hypervisor CPUID information. On Microsoft hypervisors, this will be at least `0x40000005`.

Register	Information Provided
EBX	0x7263694D—“Micr”
ECX	0x666F736F—“osof”
EDX	0x76482074—“t Hv”

Hypervisor Vendor-Neutral Interface Identification - 0x40000001

EAX contains the hypervisor interface identification signature. This determines the semantics of the leaves from `0x40000002` through `0x400000FF`.

Register	Information Provided
EAX	0x31237648—“Hv#1”
EBX	Reserved
ECX	Reserved
EDX	Reserved

Hypervisors conforming to the “Hv#1” interface also provide at least the following leaves.

Hypervisor System Identity - 0x40000002

Register	Bits	Information Provided
EAX		Build Number
EBX	31-16	Major Version
	15-0	Minor Version

Hypervisor Feature Identification - 0x40000003

EAX and EBX indicate which features are available to the partition based upon the current partition privileges.

Register	Bits	Information Provided
EAX		Corresponds to bits 31-0 of HV_PARTITION_PRIVILEGE_MASK

Register	Bits	Information Provided
EBX		Corresponds to bits 63-32 of HV_PARTITION_PRIVILEGE_MASK
ECX	4-0	Reserved
	5	Invariant Mperf is available
	6	Supervisor shadow stack is available
	7	Architectural PMU is available
	8	Exception trap intercept is available
	31-9	Reserved
EDX	0	Deprecated (previously indicated availability of the MWAIT instruction)
	1	Guest debugging support is available
	2	Performance Monitor support is available
	3	Support for physical CPU dynamic partitioning events is available
	4	Support for passing hypercall input parameter block via XMM registers is available
	5	Support for a virtual guest idle state is available
	6	Support for hypervisor sleep state is available
	7	Support for querying NUMA distances is available
	8	Support for determining timer frequencies is available
	9	Support for injecting synthetic machine checks is available
	10	Support for guest crash MSRs is available
	11	Support for debug MSRs is available
	12	Support for NPIEP is available
	13	DisableHypervisorAvailable
	14	ExtendedGvaRangesForFlushVirtualAddressListAvailable
	15	Support for returning hypercall output via XMM registers is available
	16	Reserved
	17	SintPollingModeAvailable
	18	HypercallMsrLockAvailable

Register	Bits	Information Provided
	19	Use direct synthetic timers
	20	Support for PAT register available for VSM
	21	Support for bndcfgs register available for VSM
	22	Reserved
	23	Support for synthetic time unhalted timer available
	25-24	Reserved
	26	Intel’s Last Branch Record (LBR) feature supported
	31-27	Reserved

Implementation Recommendations - 0x40000004

Indicates which behaviors the hypervisor recommends the OS implement for optimal performance.

Register	Bits	Information Provided
EAX	0	Recommend using hypercall for address space switches rather than MOV to CR3 instruction.
	1	Recommend using hypercall for local TLB flushes rather than INVLPG or MOV to CR3 instructions.
	2	Recommend using hypercall for remote TLB flushes rather than inter-processor interrupts.
	3	Recommend using MSRs for accessing APIC registers EOI, ICR and TPR rather than their memory-mapped counterparts.
	4	Recommend using the hypervisor-provided MSR to initiate a system RESET.
	5	Recommend using relaxed timing for this partition. If used, the VM should disable any watchdog timeouts that rely on the timely delivery of external interrupts.
	6	Recommend using DMA remapping.
	7	Recommend using interrupt remapping.
	8	Reserved.

Register	Bits	Information Provided
	9	Recommend deprecating AutoEOI.
	10	Recommend using SyntheticClusterIpi hypercall.
	11	Recommend using the newer ExProcessorMasks interface.
	12	Indicates that the hypervisor is nested within a Hyper-V partition.
	13	Recommend using INT for MBEC system calls.
	14	Recommend a nested hypervisor using the enlightened VMCS interface. Also indicates that additional nested enlightenments may be available (see leaf 0x4000000A).
	15	UseSyncedTimeline – Indicates the partition should consume the QueryPerformanceCounter bias provided by the root partition.
	16	Reserved
	17	UseDirectLocalFlushEntire – Indicates the guest should toggle CR4.PGE to flush the entire TLB, as this is more performant than making a hypercall.
	18	NoNonArchitecturalCoreSharing - indicates that core sharing is not possible. This can be used as an optimization to avoid the performance overhead of STIBP.
	31-19	Reserved
EBX		Recommended number of attempts to retry a spinlock failure before notifying the hypervisor about the failures. 0xFFFFFFFF indicates never notify.
ECX	6-0	ImplementedPhysicalAddressBits – Reports the physical address width (MAXPHYADDR) reported by the system's physical processors. If all bits contain 0, the feature is not supported. Note that the value reported is the actual number of physical address bits, and not the bit position used to represent that number.
	31-7	Reserved
EDX		Reserved

Hypervisor Implementation Limits - 0x40000005

Describes the scale limits supported in the current hypervisor implementation. If any value is zero, the hypervisor does not expose the corresponding information; otherwise, they have these meanings.

Register	Information Provided
EAX	The maximum number of virtual processors supported
EBX	The maximum number of logical processors supported
ECX	The maximum number of physical interrupt vectors available for interrupt remapping.
EDX	Reserved

Implementation Hardware Features - 0x40000006

Indicates which hardware-specific features have been detected and are currently in use by the hypervisor.

Register	Bits	Information Provided
EAX	0	Support for APIC overlay assist is detected and in use.
	1	Support for MSR bitmaps is detected and in use.
	2	Support for architectural performance counters is detected and in use.
	3	Support for second level address translation is detected and in use.
	4	Support for DMA remapping is detected and in use.
	5	Support for interrupt remapping is detected and in use.
	6	Indicates that a memory patrol scrubber is present in the hardware.
	7	DMA protection is in use.
	8	HPET is requested.
	9	Synthetic timers are volatile.
	13-10	The hypervisor level of the current guest - '0' if non-nested.
	14	Physical destination mode required.
	15	Reserved
	16	Support for hardware memory zeroing is present.
	17	Support for Unrestricted Guest is present.
	18	Support for resource allocation (RDT-A, PQOS-A) is present.
	19	Support for resource monitoring (RDT-M, PQOS-M) is present.

Register	Bits	Information Provided
	20	Support for guest virtual PMU is present.
	21	Support for guest virtual LBR is present.
	22	Support for guest virtual IPT is present.
	23	Support for APIC emulation is present.
	24	ACPI WDAT table is detected and in use by the hypervisor.
	31-25	Reserved
EBX		Reserved
ECX		Reserved
EDX		Reserved

Nested Hypervisor Feature Identification - 0x40000009

Describes the features exposed to the partition by the hypervisor when running nested. EAX describes access to virtual MSRs. EDX describes access to hypercalls.

Register	Bits	Information Provided
EAX	1-0	Reserved
	2	AccessSynicRegs
	3	Reserved
	4	AccessIntrCtrlRegs
	5	AccessHypercallMsrs
	6	AccessVpIndex
	11-7	Reserved
	12	AccessReenlightenmentControls
	31-13	Reserved
EBX		Reserved
ECX		Reserved
EDX	3-0	Reserved

Register	Bits	Information Provided
	4	XmmRegistersForFastHypercallAvailable
	14-5	Reserved
	15	FastHypercallOutputAvailable
	16	Reserved
	17	SintPollingModeAvailable
	31-18	Reserved

Hypervisor Nested Virtualization Features - 0x4000000A

Indicates which nested virtualization optimizations are available to a nested hypervisor.

Register	Bits	Information Provided
EAX	7-0	Enlightened VMCS version (low)
	15-8	Enlightened VMCS version (high)
	16	Reserved
	17	Indicates support for direct virtual flush hypercalls.
	18	Indicates support for the HvFlushGuestPhysicalAddressSpace and HvFlushGuestPhysicalAddressList hypercalls (on Intel platforms).
	19	Indicates support for using an enlightened MSR bitmap.
	20	Indicates support for combining virtualization exceptions in the page fault exception class.
	21	Indicates support for non-zero value of the 0x00002802 (GuestIa32DebugCtl) field in the VMCS.
	22	Indicates support for the enlightened TLB on AMD platforms. ASID flushes do not affect TLB entries derived from the NPT. Hypercalls must be used to invalidate NPT TLB entries. Also indicates support for the HvFlushGuestPhysicalAddressSpace and HvFlushGuestPhysicalAddressList hypercalls.
	31-21	Reserved

Register	Bits	Information Provided
EBX	0	Indicates support for the GuestPerfGlobalCtrl and HostPerfGlobalCtrl fields in the enlightened VMCS.
	31-1	Reserved
ECX		Reserved
EDX		Reserved

Versioning

The hypervisor version information is encoded in leaf `0x40000002`. Two version numbers are provided: the main version and the service version.

The main version includes a major and minor version number and a build number. These correspond to Microsoft Windows release numbers. The service version describes changes made to the main version.

Clients are strongly encouraged to check for hypervisor features by using CPUID leaves `0x40000003` through `0x40000005` rather than by comparing against version ranges.

Hypercall Interface

Article • 05/02/2022

The hypervisor provides a calling mechanism for guests. Such calls are referred to as hypercalls. Each hypercall defines a set of input and/or output parameters. These parameters are specified in terms of a memory-based data structure. All elements of the input and output data structures are padded to natural boundaries up to 8 bytes (that is, two-byte elements must be on two-byte boundaries and so on).

A second hypercall calling convention can optionally be used for a subset of hypercalls – in particular, those that have two or fewer input parameters and no output parameters. When using this calling convention, the input parameters are passed in general-purpose registers.

A third hypercall calling convention can optionally be used for a subset of hypercalls where the input parameter block is up to 112 bytes. When using this calling convention, the input parameters are passed in registers, including the volatile XMM registers.

Input and output data structures must both be placed in memory on an 8-byte boundary and padded to a multiple of 8 bytes in size. The values within the padding regions are ignored by the hypervisor.

For output, the hypervisor is allowed to (but not guaranteed to) overwrite padding regions. If it overwrites padding regions, it will write zeros.

Hypercall Classes

There are two classes of hypercalls: simple and rep (short for "repeat"). A simple hypercall performs a single operation and has a fixed-size set of input and output parameters. A rep hypercall acts like a series of simple hypercalls. In addition to a fixed-size set of input and output parameters, rep hypercalls involve a list of fixed-size input and/or output elements.

When a caller initially invokes a rep hypercall, it specifies a rep count that indicates the number of elements in the input and/or output parameter list. Callers also specify a rep start index that indicates the next input and/or output element that should be consumed. The hypervisor processes rep parameters in list order – that is, by increasing element index.

For subsequent invocations of the rep hypercall, the rep start index indicates how many elements have been completed – and, in conjunction with the rep count value – how

many elements are left. For example, if a caller specifies a rep count of 25, and only 20 iterations are completed within the time constraints, the hypercall returns control back to the calling virtual processor after updating the rep start index to 20. When the hypercall is re-executed, the hypervisor will resume at element 20 and complete the remaining 5 elements.

If an error is encountered when processing an element, an appropriate status code is provided along with a reps completed count, indicating the number of elements that were successfully processed before the error was encountered. Assuming the specified hypercall control word is valid (see the following) and the input / output parameter lists are accessible, the hypervisor is guaranteed to attempt at least one rep, but it is not required to process the entire list before returning control back to the caller.

Hypercall Continuation

A hypercall can be thought of as a complex instruction that takes many cycles. The hypervisor attempts to limit hypercall execution to 50µs or less before returning control to the virtual processor that invoked the hypercall. Some hypercall operations are sufficiently complex that a 50µs guarantee is difficult to make. The hypervisor therefore relies on a hypercall continuation mechanism for some hypercalls – including all rep hypercall forms.

The hypercall continuation mechanism is mostly transparent to the caller. If a hypercall is not able to complete within the prescribed time limit, control is returned back to the caller, but the instruction pointer is not advanced past the instruction that invoked the hypercall. This allows pending interrupts to be handled and other virtual processors to be scheduled. When the original calling thread resumes execution, it will re-execute the hypercall instruction and make forward progress toward completing the operation.

Most simple hypercalls are guaranteed to complete within the prescribed time limit. However, a small number of simple hypercalls might require more time. These hypercalls use hypercall continuation in a similar manner to rep hypercalls. In such cases, the operation involves two or more internal states. The first invocation places the object (for example, the partition or virtual processor) into one state, and after repeated invocations, the state finally transitions to a terminal state. For each hypercall that follows this pattern, the visible side effects of intermediate internal states is described.

Hypercall Atomicity and Ordering

Except where noted, the action performed by a hypercall is atomic both with respect to all other guest operations (for example, instructions executed within a guest) and all

other hypercalls being executed on the system. A simple hypercall performs a single atomic action; a rep hypercall performs multiple, independent atomic actions.

Simple hypercalls that use hypercall continuation may involve multiple internal states that are externally visible. Such calls comprise multiple atomic operations.

Each hypercall action may read input parameters and/or write results. The inputs to each action can be read at any granularity and at any time after the hypercall is made and before the action is executed. The results (that is, the output parameters) associated with each action may be written at any granularity and at any time after the action is executed and before the hypercall returns.

The guest must avoid the examination and/or manipulation of any input or output parameters related to an executing hypercall. While a virtual processor executing a hypercall will be incapable of doing so (as its guest execution is suspended until the hypercall returns), there is nothing to prevent other virtual processors from doing so. Guests behaving in this manner may crash or cause corruption within their partition.

Legal Hypercall Environments

Hypercalls can be invoked only from the most privileged guest processor mode. On x64 platfoms, this means protected mode with a current privilege level (CPL) of zero. Although real-mode code runs with an effective CPL of zero, hypercalls are not allowed in real mode. An attempt to invoke a hypercall within an illegal processor mode will generate a #UD (undefined operation) exception.

All hypercalls should be invoked through the architecturally-defined hypercall interface (see below). An attempt to invoke a hypercall by any other means (for example, copying the code from the hypercall code page to an alternate location and executing it from there) might result in an undefined operation (#UD) exception. The hypervisor is not guaranteed to deliver this exception.

Alignment Requirements

Callers must specify the 64-bit guest physical address (GPA) of the input and/or output parameters. GPA pointers must by 8-byte aligned. If the hypercall involves no input or output parameters, the hypervisor ignores the corresponding GPA pointer.

The input and output parameter lists cannot overlap or cross page boundaries. Hypercall input and output pages are expected to be GPA pages and not “overlay” pages. If the virtual processor writes the input parameters to an overlay page and

specifies a GPA within this page, hypervisor access to the input parameter list is undefined.

The hypervisor will validate that the calling partition can read from the input page before executing the requested hypercall. This validation consists of two checks: the specified GPA is mapped and the GPA is marked readable. If either of these tests fails, the hypervisor generates a memory intercept message. For hypercalls that have output parameters, the hypervisor will validate that the partition can be write to the output page. This validation consists of two checks: the specified GPA is mapped and the GPA is marked writable.

Hypercall Inputs

Callers specify a hypercall by a 64-bit value called a hypercall input value. It is formatted as follows:

Field	Bits	Information Provided
Call Code	15-0	Specifies which hypercall is requested
Fast	16	Specifies whether the hypercall uses the register-based calling convention: 0 = memory-based, 1 = register-based
Variable header size	26-17	The size of a variable header, in QWORDS.
RsvdZ	30-27	Must be zero
Is Nested	31	Specifies the hypercall should be handled by the L0 hypervisor in a nested environment.
Rep Count	43-32	Total number of reps (for rep call, must be zero otherwise)
RsvdZ	47-44	Must be zero
Rep Start Index	59-48	Starting index (for rep call, must be zero otherwise)
RsvdZ	63-60	Must be zero

For rep hypercalls, the rep count field indicates the total number of reps. The rep start index indicates the particular repetition relative to the start of the list (zero indicates that

the first element in the list is to be processed). Therefore, the rep count value must always be greater than the rep start index.

Register mapping for hypercall inputs when the Fast flag is zero:

x64	x86	Information Provided
RCX	EDX:EAX	Hypercall Input Value
RDX	EBX:ECX	Input Parameters GPA
R8	EDI:ESI	Output Parameters GPA

The hypercall input value is passed in registers along with a GPA that points to the input and output parameters.

On x64, the register mappings depend on whether the caller is running in 32-bit (x86) or 64-bit (x64) mode. The hypervisor determines the caller's mode based on the value of EFER.LMA and CS.L. If both of these flags are set, the caller is assumed to be a 64-bit caller.

Register mapping for hypercall inputs when the Fast flag is one:

x64	x86	Information Provided
RCX	EDX:EAX	Hypercall Input Value
RDX	EBX:ECX	Input Parameter
R8	EDI:ESI	Output Parameter

The hypercall input value is passed in registers along with the input parameters.

Variable Sized Hypercall Input Headers

Most hypercall input headers have fixed size. The amount of header data being passed from the guest to the hypervisor is therefore implicitly specified by the hypercall code and need not be specified separately. However, some hypercalls require a variable amount of header data. These hypercalls typically have a fixed size input header and additional header input that is of variable size.

A variable sized header is similar to a fixed hypercall input (aligned to 8 bytes and sized to a multiple of 8 bytes). The caller must specify how much data it is providing as input headers. This size is provided as part of the hypercall input value (see "Variable header size" in table above).

Since the fixed header size is implicit, instead of supplying the total header size, only the variable portion is supplied in the input controls:

```
Variable Header Bytes = {Total Header Bytes - sizeof(Fixed Header)} rounded
up to nearest multiple of 8

Variable HeaderSize = Variable Header Bytes / 8
```

It is illegal to specify a non-zero variable header size for a hypercall that is not explicitly documented as accepting variable sized input headers. In such a case the hypercall will result in a return code of `HV_STATUS_INVALID_HYPERCALL_INPUT`.

It is possible that for a given invocation of a hypercall that does accept variable sized input headers that all the header input fits entirely within the fixed size header. In such cases the variable sized input header is zero-sized and the corresponding bits in the hypercall input should be set to zero.

In all other regards, hypercalls accepting variable sized input headers are otherwise similar to fixed size input header hypercalls with regards to calling conventions. It is also possible for a variable sized header hypercall to additionally support rep semantics. In such a case the rep elements lie after the header in the usual fashion, except that the header's total size includes both the fixed and variable portions. All other rules remain the same, e.g. the first rep element must be 8 byte aligned.

XMM Fast Hypercall Input

On x64 platforms, the hypervisor supports the use of XMM fast hypercalls, which allows some hypercalls to take advantage of the improved performance of the fast hypercall interface even though they require more than two input parameters. The XMM fast hypercall interface uses six XMM registers to allow the caller to pass an input parameter block up to 112 bytes in size.

Availability of the XMM fast hypercall interface is indicated via the “Hypervisor Feature Identification” CPUID Leaf (0x40000003):

- Bit 4: support for passing hypercall input via XMM registers is available.

Note that there is a separate flag to indicate support for XMM fast output. Any attempt to use this interface when the hypervisor does not indicate availability will result in a #UD fault.

Register Mapping (Input Only)

x64	x86	Information Provided
RCX	EDX:EAX	Hypercall Input Value
RDX	EBX:ECX	Input Parameter Block
R8	EDI:ESI	Input Parameter Block
XMM0	XMM0	Input Parameter Block
XMM1	XMM1	Input Parameter Block
XMM2	XMM2	Input Parameter Block
XMM3	XMM3	Input Parameter Block
XMM4	XMM4	Input Parameter Block
XMM5	XMM5	Input Parameter Block

The hypercall input value is passed in registers along with the input parameters. The register mappings depend on whether the caller is running in 32-bit (x86) or 64-bit (x64) mode. The hypervisor determines the caller's mode based on the value of EFER.LMA and CS.L. If both of these flags are set, the caller is assumed to be a 64-bit caller. If the input parameter block is smaller than 112 bytes, any extra bytes in the registers are ignored.

Hypercall Outputs

All hypercalls return a 64-bit value called a hypercall result value. It is formatted as follows:

Field	Bits	Comment
Result	15-0	`HV_STATUS` code indicating success or failure
Rsvd	31-16	Callers should ignore the value in these bits
Reps completed	43-32	Number of reps successfully completed
RsvdZ	63-40	Callers should ignore the value in these bits

For rep hypercalls, the reps complete field is the total number of reps complete and not relative to the rep start index. For example, if the caller specified a rep start index of 5, and a rep count of 10, the reps complete field would indicate 10 upon successful completion.

The hypercall result value is passed back in registers. The register mapping depends on whether the caller is running in 32-bit (x86) or 64-bit (x64) mode (see above). The register mapping for hypercall outputs is as follows:

x64	x86	Information Provided
RAX	EDX:EAX	Hypercall Result Value

XMM Fast Hypercall Output

Similar to how the hypervisor supports XMM fast hypercall inputs, the same registers can be shared to return output. This is only supported on x64 platforms.

The ability to return output via XMM registers is indicated via the “Hypervisor Feature Identification” CPUID Leaf (0x40000003):

- Bit 15: support for returning hypercall output via XMM registers is available.

Note that there is a separate flag to indicate support for XMM fast input. Any attempt to use this interface when the hypervisor does not indicate availability will result in a #UD fault.

Register Mapping (Input and Output)

Registers that are not being used to pass input parameters can be used to return output. In other words, if the input parameter block is smaller than 112 bytes (rounded up to the nearest 16 byte aligned chunk), the remaining registers will return hypercall output.

x64	Information Provided
RDX	Input or Output Block
R8	Input or Output Block
XMM0	Input or Output Block
XMM1	Input or Output Block
XMM2	Input or Output Block
XMM3	Input or Output Block
XMM4	Input or Output Block
XMM5	Input or Output Block

For example, if the input parameter block is 20 bytes in size, the hypervisor would ignore the following 12 bytes. The remaining 80 bytes would contain hypercall output (if applicable).

Volatile Registers

Hypercalls will only modify the specified register values under the following conditions:

1. RAX (x64) and EDX:EAX (x86) are always overwritten with the hypercall result value and output parameters, if any.
2. Rep hypercalls will modify RCX (x64) and EDX:EAX (x86) with the new rep start index.
3. HvCallSetVpRegisters can modify any registers that are supported with that hypercall.
4. RDX, R8, and XMM0 through XMM5, when used for fast hypercall input, remain unmodified. However, registers used for fast hypercall output can be modified, including RDX, R8, and XMM0 through XMM5. Hyper-V will only modify these registers for fast hypercall output, which is limited to x64.

Hypercall Restrictions

Hypercalls may have restrictions associated with them for them to perform their intended function. If all restrictions are not met, the hypercall will terminate with an appropriate error. The following restrictions will be listed, if any apply:

- The calling partition must possess a particular privilege
- The partition being acted upon must be in a particular state (e.g. “Active”)

Hypercall Status Codes

Each hypercall is documented as returning an output value that contains several fields. A status value field (of type `HV_STATUS`) is used to indicate whether the call succeeded or failed.

Output Parameter Validity on Failed Hypercalls

Unless explicitly stated otherwise, when a hypercall fails (that is, the result field of the hypercall result value contains a value other than `HV_STATUS_SUCCESS`), the content of all output parameters are indeterminate and should not be examined by the caller. Only

when the hypercall succeeds, will all appropriate output parameters contain valid, expected results.

Ordering of Error Conditions

The order in which error conditions are detected and reported by the hypervisor is undefined. In other words, if multiple errors exist, the hypervisor must choose which error condition to report. Priority should be given to those error codes offering greater security, the intent being to prevent the hypervisor from revealing information to callers lacking sufficient privilege. For example, the status code `HV_STATUS_ACCESS_DENIED` is the preferred status code over one that would reveal some context or state information purely based upon privilege.

Common Hypercall Status Codes

Several result codes are common to all hypercalls and are therefore not documented for each hypercall individually. These include the following:

Status code	Error condition
`HV_STATUS_SUCCESS`	The call succeeded.
`HV_STATUS_INVALID_HYPERCALL_CODE`	The hypercall code is not recognized.
`HV_STATUS_INVALID_HYPERCALL_INPUT`	The rep count is incorrect (for example, a non-zero rep count is passed to a non-rep call or a zero rep count is passed to a rep call).
	The rep start index is not less than the rep count.
	A reserved bit in the specified hypercall input value is non-zero.
`HV_STATUS_INVALID_ALIGNMENT`	The specified input or output GPA pointer is not aligned to 8 bytes.
	The specified input or output parameter lists spans pages.
	The input or output GPA pointer is not within the bounds of the GPA space.

The return code `HV_STATUS_SUCCESS` indicates that no error condition was detected.

Reporting the Guest OS Identity

The guest OS running within the partition must identify itself to the hypervisor by writing its signature and version to an MSR (`HV_X64_MSR_GUEST_OS_ID`) before it can invoke hypercalls. This MSR is partition-wide and is shared among all virtual processors.

This register's value is initially zero. A non-zero value must be written to the Guest OS ID MSR before the hypercall code page can be enabled (see Establishing the Hypercall Interface). If this register is subsequently zeroed, the hypercall code page will be disabled.

```c
#define HV_X64_MSR_GUEST_OS_ID 0x40000000
```

Guest OS Identity for proprietary Operating Systems

The following is the recommended encoding for this MSR. Some fields may not apply for some guest OSs.

Bits	Field	Description
15:0	Build Number	Indicates the build number of the OS
23:16	Service Version	Indicates the service version (for example, "service pack" number)
31:24	Minor Version	Indicates the minor version of the OS
39:32	Major Version	Indicates the major version of the OS
47:40	OS ID	Indicates the OS variant. Encoding is unique to the vendor. Microsoft operating systems are encoded as follows: 0=Undefined, 1=MS-DOS®, 2=Windows® 3.x, 3=Windows® 9x, 4=Windows® NT (and derivatives), 5=Windows® CE
62:48	Vendor ID	Indicates the guest OS vendor. A value of 0 is reserved. See list of vendors below.
63	OS Type	Indicates the OS types. A value of 0 indicates a proprietary, closed source OS. A value of 1 indicates an open source OS.

Vendor values are allocated by Microsoft. To request a new vendor, please file an issue on the GitHub virtualization documentation repository (https://aka.ms/VirtualizationDocumentationIssuesTLFS).

Vendor	Value
Microsoft	0x0001
HPE	0x0002
LANCOM	0x0200

Guest OS Identity MSR for Open Source Operating Systems

The following encoding is offered as guidance for open source operating system vendors intending to conform to this specification. It is suggested that open source operating systems adapt the following convention.

Bits	Field	Description
15:0	Build Number	Additional Information
47:16	Version	Upstream kernel version information.
55:48	OS ID	Additional vendor information
62:56	OS Type	OS type (e.g., Linux, FreeBSD, etc.). See list of known OS types below
63	Open Source	A value of 1 indicates an open source OS.

OS Type values are allocated by Microsoft. To request a new OS Type, please file an issue on the GitHub virtualization documentation repository (https://aka.ms/VirtualizationDocumentationIssuesTLFS).

OS Type	Value
Linux	0x1
FreeBSD	0x2
Xen	0x3
Illumos	0x4

Establishing the Hypercall Interface

Hypercalls are invoked by using a special opcode. Because this opcode differs among virtualization implementations, it is necessary for the hypervisor to abstract this difference. This is done through a special hypercall page. This page is provided by the

hypervisor and appears within the guest's GPA space. The guest is required to specify the location of the page by programming the Guest Hypercall MSR.

```C
#define HV_X64_MSR_HYPERCALL 0x40000001
```

Bits	Description	Attributes
63:12	Hypercall GPFN - Indicates the Guest Physical Page Number of the hypercall page	Read/write
11:2	RsvdP. Bits should be ignored on reads and preserved on writes.	Reserved
1	Locked. Indicates if the MSR is immutable. If set, this MSR is locked thereby preventing the relocation of the hypercall page. Once set, only a system reset can clear the bit.	Read/write
0	Enable hypercall page	Read/write

The hypercall page can be placed anywhere within the guest's GPA space, but must be page-aligned. If the guest attempts to move the hypercall page beyond the bounds of the GPA space, a #GP fault will result when the MSR is written.

This MSR is a partition-wide MSR. In other words, it is shared by all virtual processors in the partition. If one virtual processor successfully writes to the MSR, another virtual processor will read the same value.

Before the hypercall page is enabled, the guest OS must report its identity by writing its version signature to a separate MSR (HV_X64_MSR_GUEST_OS_ID). If no guest OS identity has been specified, attempts to enable the hypercall will fail. The enable bit will remain zero even if a one is written to it. Furthermore, if the guest OS identity is cleared to zero after the hypercall page has been enabled, it will become disabled.

The hypercall page appears as an "overlay" to the GPA space; that is, it covers whatever else is mapped to the GPA range. Its contents are readable and executable by the guest. Attempts to write to the hypercall page will result in a protection (#GP) exception. After the hypercall page has been enabled, invoking a hypercall simply involves a call to the start of the page.

The following is a detailed list of the steps involved in establishing the hypercall page:

1. The guest reads CPUID leaf 1 and determines whether a hypervisor is present by checking bit 31 of register ECX.

2. The guest reads CPUID leaf 0x40000000 to determine the maximum hypervisor CPUID leaf (returned in register EAX) and CPUID leaf 0x40000001 to determine the interface signature (returned in register EAX). It verifies that the maximum leaf value is at least 0x40000005 and that the interface signature is equal to "Hv#1". This signature implies that `HV_X64_MSR_GUEST_OS_ID`, `HV_X64_MSR_HYPERCALL` and `HV_X64_MSR_VP_INDEX` are implemented.
3. The guest writes its OS identity into the MSR `HV_X64_MSR_GUEST_OS_ID` if that register is zero.
4. The guest reads the Hypercall MSR (`HV_X64_MSR_HYPERCALL`).
5. The guest checks the Enable Hypercall Page bit. If it is set, the interface is already active, and steps 6 and 7 should be omitted.
6. The guest finds a page within its GPA space, preferably one that is not occupied by RAM, MMIO, and so on. If the page is occupied, the guest should avoid using the underlying page for other purposes.
7. The guest writes a new value to the Hypercall MSR (`HV_X64_MSR_HYPERCALL`) that includes the GPA from step 6 and sets the Enable Hypercall Page bit to enable the interface.
8. The guest creates an executable VA mapping to the hypercall page GPA.
9. The guest consults CPUID leaf 0x40000003 to determine which hypervisor facilities are available to it. After the interface has been established, the guest can initiate a hypercall. To do so, it populates the registers per the hypercall protocol and issues a CALL to the beginning of the hypercall page. The guest should assume the hypercall page performs the equivalent of a near return (0xC3) to return to the caller. As such, the hypercall must be invoked with a valid stack.

Extended Hypercall Interface

Hypercalls with call codes above 0x8000 are known as extended hypercalls. Extended hypercalls use the same calling convention as normal hypercalls and appear identical from a guest VM's perspective. Extended hypercalls are internally handled differently within the Hyper-V hypervisor.

Extended hypercall capabilities can be queried with HvExtCallQueryCapabilities.

Hypercall Reference

Article • 03/17/2023

The following table lists supported hypercalls by call code.

Call Code	Type	Hypercall
0x0001	Simple	HvCallSwitchVirtualAddressSpace
0x0002	Simple	HvCallFlushVirtualAddressSpace
0x0003	Rep	HvCallFlushVirtualAddressList
0x0008	Simple	HvCallNotifyLongSpinWait
0x000b	Simple	HvCallSendSyntheticClusterIpi
0x000c	Rep	HvCallModifyVtlProtectionMask
0x000d	Simple	HvCallEnablePartitionVtl
0x000f	Simple	HvCallEnableVpVtl
0x0011	Simple	HvCallVtlCall
0x0012	Simple	HvCallVtlReturn
0x0013	Simple	HvCallFlushVirtualAddressSpaceEx
0x0014	Rep	HvCallFlushVirtualAddressListEx
0x0015	Simple	HvCallSendSyntheticClusterIpiEx
0x0050	Rep	HvCallGetVpRegisters
0x0051	Rep	HvCallSetVpRegisters
0x005C	Simple	HvCallPostMessage
0x005D	Simple	HvCallSignalEvent
0x007e	Simple	HvCallRetargetDeviceInterrupt
0x0099	Simple	HvCallStartVirtualProcessor
0x009A	Rep	HvCallGetVpIndexFromApicId
0x00AF	Simple	HvCallFlushGuestPhysicalAddressSpace
0x00B0	Rep	HvCallFlushGuestPhysicalAddressList

The following table lists supported extended hypercalls by call code.

Call Code	Type	Hypercall
0x8001	Simple	HvExtCallQueryCapabilities
0x8002	Simple	HvExtCallGetBootZeroedMemory
0x8003	Simple	HvExtCallMemoryHeatHint
0x8004	Simple	HvExtCallEpfSetup
0x8006	Simple	HvExtCallMemoryHeatHintAsync

Partitions

Article • 05/26/2021

The hypervisor supports isolation in terms of a partition. A partition is a logical unit of isolation, supported by the hypervisor, in which operating systems execute.

Partition Privilege Flags

Each partition has a set of privileges assigned by the hypervisor. Privileges control access to synthetic MSRs or hypercalls.

A partition can query its privileges through the "Hypervisor Feature Identification" CPUID Leaf (0x40000003). See HV_PARTITION_PRIVILEGE_MASK for a description of all privileges.

Partition Crash Enlightenment

The hypervisor provides guest partitions with a crash enlightenment facility. This interface allows the operating system running in a guest partition the option of providing forensic information about fatal OS conditions to the hypervisor as part of its crash dump procedure. Options include preserving the contents of the guest crash parameter MSRs and specifying a crash message. The hypervisor then makes this information available to the root partition for logging. This mechanism allows the virtualization host administrator to gather information about the guest OS crash event without needing to inspect persistent storage attached to the guest partition for crash dump or core dump information that may be stored there by the crashing guest OS.

The availability of this mechanism is indicated via `CPUID.0x400003.EDX:10`, the GuestCrashMsrsAvailable flag; refer to feature discovery.

Guest Crash Enlightenment Interface

The guest crash enlightenment interface is provided through six synthetic MSRs, as defined below.

```C
#define HV_X64_MSR_CRASH_P0 0x40000100
#define HV_X64_MSR_CRASH_P1 0x40000101
#define HV_X64_MSR_CRASH_P2 0x40000102
#define HV_X64_MSR_CRASH_P3 0x40000103
```

```
#define HV_X64_MSR_CRASH_P4 0x40000104
#define HV_X64_MSR_CRASH_CTL 0x40000105
```

Guest Crash Control MSR

The guest crash control MSR HV_X64_MSR_CRASH_CTL may be used by guest partitions to determine the hypervisor's guest crash capabilities, and to invoke the specified action to take. The HV_CRASH_CTL_REG_CONTENTS data structure defines the contents of the MSR.

Determining Guest Crash Capabilities

To determine the guest crash capabilities, guest partitions may read the HV_X64_MSR_CRASH_CTL register. The supported set of actions and capabilities supported by the hypervisor is reported.

Invoking Guest Crash Capabilities

To invoke a supported hypervisor guest crash action, a guest partition writes to the HV_X64_MSR_CRASH_CTL register, specifying the desired action. Two variations are supported: CrashNotify by itself, and CrashMessage in combination with CrashNotify. For each occurrence of a guest crash, at most a single write to MSR HV_X64_MSR_CRASH_CTL should be performed, specifying one of the two variations.

Guest Crash Action	Description
CrashMessage	This action is used in combination with CrashNotify to specify a crash message to the hypervisor. When selected, the values of P3 and P4 are treated as the location and size of the message. HV_X64_MSR_CRASH_P3 is the guest physical address of the message, and HV_X64_MSR_CRASH_P4 is the length in bytes of the message (maximum of 4096 bytes).
CrashNotify	This action indicates to the hypervisor that the guest partition has completed writing the desired data into the guest crash parameter MSRs (i.e., P0 thru P4), and the hypervisor should proceed with logging the contents of these MSRs.

Virtual Processors

Article • 03/17/2023

Each partition may have zero or more virtual processors.

Virtual Processor Indices

A virtual processor is identified by a tuple composed of its partition ID and its processor index. The processor index is assigned to the virtual processor when it is created, and it is unchanged through the lifetime of the virtual processor.

A special value HV_ANY_VP can be used in certain situations to specify "any virtual processor". A value of HV_VP_INDEX_SELF can be used to specify one's own VP index.

```c
typedef UINT32 HV_VP_INDEX;

#define HV_ANY_VP ((HV_VP_INDEX)-1)

#define HV_VP_INDEX_SELF ((HV_VP_INDEX)-2)
```

A virtual processor's ID can be retrieved by the guest through a hypervisor-defined MSR (model-specific register) HV_X64_MSR_VP_INDEX.

```c
#define HV_X64_MSR_VP_INDEX 0x40000002
```

Virtual Processor Idle Sleep State

Virtual processors may be placed in a virtual idle processor power state, or processor sleep state. This enhanced virtual idle state allows a virtual processor that is placed into a low power idle state to be woken with the arrival of an interrupt even when the interrupt is masked on the virtual processor. In other words, the virtual idle state allows the operating system in the guest partition to take advantage of processor power saving techniques in the OS that would otherwise be unavailable when running in a guest partition.

A partition which possesses the AccessGuestIdleMsr privilege may trigger entry into the virtual processor idle sleep state through a read to the hypervisor-defined MSR

`HV_X64_MSR_GUEST_IDLE`. The virtual processor will be woken when an interrupt arrives, regardless of whether the interrupt is enabled on the virtual processor or not.

Virtual Processor Assist Page

The hypervisor provides a page per virtual processor which is overlaid on the guest GPA space. This page can be used for bi-directional communication between a guest VP and the hypervisor. The guest OS has read/write access to this virtual VP assist page.

A guest specifies the location of the overlay page (in GPA space) by writing to the Virtual VP Assist MSR (0x40000073). The format of the Virtual VP Assist Page MSR is as follows:

Bits	Field	Description
0	Enable	Enables the VP assist page
11:1	RsvdP	Reserved
63:12	Page PFN	Virtual VP Assist Page PFN

Virtual Processor Run Time Register

The hypervisor's scheduler internally tracks how much time each virtual processor consumes in executing code. The time tracked is a combination of the time the virtual processor consumes running guest code, and the time the associated logical processor spends running hypervisor code on behalf of that guest. This cumulative time is accessible through the 64-bit read-only HV_X64_MSR_VP_RUNTIME hypervisor MSR. The time quantity is measured in 100ns units.

Non-Privileged Instruction Execution Prevention (NPIEP)

Non-Privileged Instruction Execution (NPIEP) is a feature that limits the use of certain instructions by user-mode code. Specifically, when enabled, this feature can block the execution of the SIDT, SGDT, SLDT, and STR instructions. Execution of these instructions results in a #GP fault.

This feature must be configured on a per-VP basis using HV_X64_MSR_NPIEP_CONFIG_CONTENTS.

Guest Spinlocks

A typical multiprocessor-capable operating system uses locks for enforcing atomicity of certain operations. When running such an operating system inside a virtual machine controlled by the hypervisor these critical sections protected by locks can be extended by intercepts generated by the critical section code. The critical section code may also be preempted by the hypervisor scheduler. Although the hypervisor attempts to prevent such preemptions, they can occur. Consequently, other lock contenders could end up spinning until the lock holder is re-scheduled again and therefore significantly extend the spinlock acquisition time.

The hypervisor indicates to the guest OS the number of times a spinlock acquisition should be attempted before indicating an excessive spin situation to the hypervisor. This count is returned in CPUID leaf 0x40000004. A value of 0 indicates that the guest OS should not notify the hypervisor about long spinlock acquisition.

The HvCallNotifyLongSpinWait hypercall provides an interface for enlightened guests to improve the statistical fairness property of a lock for multiprocessor virtual machines. The guest should make this notification on every multiple of the recommended count returned by CPUID leaf 0x40000004. Through this hypercall, a guest notifies the hypervisor of a long spinlock acquisition. This allows the hypervisor to make better scheduling decisions.

Virtual MMU

Article • 05/26/2021

The virtual machine interface exposed by each partition includes a memory management unit (MMU). The virtual MMU exposed by hypervisor partitions is generally compatible with existing MMUs.

Virtual MMU Overview

Virtual processors expose virtual memory and a virtual TLB (translation look-aside buffer), which caches translations from virtual addresses to (guest) physical addresses. As with the TLB on a logical processor, the virtual TLB is a non-coherent cache, and this non-coherence is visible to guests. The hypervisor exposes operations to flush the TLB. Guests can use these operations to remove potentially inconsistent entries and make virtual address translations predictable.

Compatibility

The virtual MMU exposed by the hypervisor is generally compatible with the physical MMU found within an x64 processor. The following guest-observable differences exist:

- The CR3.PWT and CR3.PCD bits may not be supported in some hypervisor implementations. On such implementations, any attempt by the guest to set these flags through a MOV to CR3 instruction or a task gate switch will be ignored. Attempts to set these bits programmatically through HvSetVpRegisters or HvSwitchVirtualAddressSpace may result in an error.
- The PWT and PCD bits within a leaf page table entry (for example, a PTE for 4-K pages and a PDE for large pages) specify the cacheability of the page being mapped. The PAT, PWT, and PCD bits within non-leaf page table entries indicate the cacheability of the next page table in the hierarchy. Some hypervisor implementations may not support these flags. On such implementations, all page table accesses performed by the hypervisor are done by using write-back cache attributes. This affects, in particular, accessed and dirty bits written to the page table entries. If the guest sets the PAT, PWT, or PCD bits within non-leaf page table entries, an “unsupported feature” message may be generated when a virtual processor accesses a page that is mapped by that page table.
- The CR0.CD (cache disable) bit may not be supported in some hypervisor implementations. On such implementations, the CR0.CD bit must be set to 0. Any attempt by the guest to set this flag through a MOV to CR0 instruction will be

ignored. Attempts to set this bit programmatically through HvSetVpRegisters will result in an error.
- The PAT (page address type) MSR is a per-VP register. However, when all the virtual processors in a partition set the PAT MSR to the same value, the new effect becomes a partition-wide effect.
- For reasons of security and isolation, the INVD instruction will be virtualized to act like a WBINVD instruction, with some differences. For security purposes, CLFLUSH should be used instead.

Legacy TLB Management Operations

The x64 architecture provides several ways to manage the processor's TLBs. The following mechanisms are virtualized by the hypervisor:

- The INVLPG instruction invalidates the translation for a single page from the processor's TLB. If the specified virtual address was originally mapped as a 4-K page, the translation for this page is removed from the TLB. If the specified virtual address was originally mapped as a "large page" (either 2 MB or 4 MB, depending on the MMU mode), the translation for the entire large page is removed from the TLB. The INVLPG instruction flushes both global and non-global translations. Global translations are defined as those which have the "global" bit set within the page table entry.
- The MOV to CR3 instruction and task switches that modify CR3 invalidate translations for all non-global pages within the processor's TLB.
- A MOV to CR4 instruction that modifies the CR4.PGE (global page enable) bit, the CR4.PSE (page size extensions) bit, or CR4.PAE (page address extensions) bit invalidates all translations (global and non-global) within the processor's TLB.

Note that all of these invalidation operations affect only one processor. To invalidate translations on other processors, software must use a software-based "TLB shoot-down" mechanism (typically implemented by using inter-process interrupts).

Virtual TLB Enhancements

In addition to supporting the legacy TLB management mechanisms described earlier, the hypervisor also supports a set of enhancements that enable a guest to manage the virtual TLB more efficiently. These enhanced operations can be used interchangeably with legacy TLB management operations.

The hypervisor supports the following hypercalls to invalidate TLBs:

Hypercall	Description
HvCallFlushVirtualAddressSpace	Invalidates all virtual TLB entries that belong to a specified address space.
HvCallFlushVirtualAddressSpaceEx	Similar to HvCallFlushVirtualAddressSpace, takes a sparse VP set as input.
HvCallFlushVirtualAddressList	Invalidates a portion of the specified address space.
HvCallFlushVirtualAddressListEx	Similar to HvCallFlushVirtualAddressList, takes a sparse VP set as input.

On some systems (those with sufficient virtualization support in hardware), the legacy TLB management instructions may be faster for local or remote (cross-processor) TLB invalidation. Guests who are interested in optimal performance should use the CPUID leaf 0x40000004 to determine which behaviors to implement using hypercalls:

- UseHypercallForAddressSpaceSwitch: If this flag is set, the caller should assume that it's faster to use HvCallSwitchAddressSpace to switch between address spaces. If this flag is clear, a MOV to CR3 instruction is recommended.
- UseHypercallForLocalFlush: If this flag is set, the caller should assume that it's faster to use hypercalls (as opposed to INVLPG or MOV to CR3) to flush one or more pages from the virtual TLB.
- UseHypercallForRemoteFlushAndLocalFlushEntire: If this flag is set, the caller should assume that it's faster to use hypercalls (as opposed to using guest-generated inter-processor interrupts) to flush one or more pages from the virtual TLB.

Memory Cache Control Overview

The hypervisor supports guest-defined cacheability settings for pages mapped within the guest's GVA space. For a detailed description of available cacheability settings and their meanings, refer to the Intel or AMD documentation.

When a virtual processor accesses a page through its GVA space, the hypervisor honors the cache attribute bits (PAT, PWT, and PCD) within the guest page table entry used to map the page. These three bits are used as an index into the partition's PAT (page address type) register to look up the final cacheability setting for the page.

Pages accessed directly through the GPA space (for example, when paging is disabled because CR0.PG is cleared) use a cacheability defined by the MTRRs. If the hypervisor implementation doesn't support virtual MTRRs, WB cacheability is assumed.

Mixing Cache Types between a Partition and the Hypervisor

Guests should be aware that some pages within its GPA space may be accessed by the hypervisor. The following list, while not exhaustive, provides several examples:

- page that contains input or output parameters for a hypercall
- All overlay pages including the hypercall page, SynIC SIEF and SIM pages, and stats pages

The hypervisor always performs accesses to hypercall parameters and overlay pages by using the WB cacheability setting.

Virtual Interrupt Controller

Article • 05/26/2021

The hypervisor virtualizes interrupt delivery to virtual processors. This is done through the use of a synthetic interrupt controller (SynIC) which is an extension of a virtualized local APIC; that is, each virtual processor has a local APIC instance with the SynIC extensions. These extensions provide a simple inter-partition communication mechanism which is described in the following chapter. Interrupts delivered to a partition fall into two categories: external and internal. External interrupts originate from other partitions or devices, and internal interrupts originate from within the partition itself.

External interrupts are generated in the following situations:

- A physical hardware device generates a hardware interrupt.
- A parent partition asserts a virtual interrupt (typically in the process of emulating a hardware device).
- The hypervisor delivers a message (for example, due to an intercept) to a partition.
- Another partition posts a message.
- Another partition signals an event.

Internal interrupts are generated in the following situations:

- A virtual processor accesses the APIC interrupt command register (ICR).
- A synthetic timer expires.

Local APIC

The SynIC is a superset of a local APIC. The interface to this APIC is given by a set of 32-bit memory mapped registers. This local APIC (including the behavior of the memory mapped registers) is generally compatible with the local APIC on P4/Xeon systems as described in Intel’s and AMD's documentation.

The hypervisor’s local APIC virtualization may deviate from physical APIC operation in the following minor ways:

- On physical systems, the IA32_APIC_BASE MSR can be different for each processor in the system. The hypervisor may require that this MSR contains the same value for all virtual processors within a partition. As such, this MSR may be treated as a partition-wide value. If a virtual processor modifies this register, the value may effectively propagate to all virtual processors within the partition.

- The IA32_APIC_BASE MSR defines a "global enable" bit for enabling or disabling the APIC. The virtualized APIC may always be enabled. If so, this bit will always be set to 1.
- The hypervisor's local APIC may not be able to generate virtual SMIs (system management interrupts).
- If multiple virtual processors within a partition are assigned identical APIC IDs, behavior of targeted interrupt delivery is boundedly undefined. That is, the hypervisor is free to deliver the interrupt to just one virtual processor, all virtual processors with the specified APIC ID, or no virtual processors. This situation is considered a guest programming error.
- Some of the memory mapped APIC registers may be accessed by way of virtual MSRs.
- The hypervisor may not allow a guest to modify its APIC IDs.

The remaining parts of this section describe only those aspects of SynIC functionality that are extensions of the local APIC.

Local APIC MSR Accesses

The hypervisor provides accelerated MSR access to high usage memory mapped APIC registers. These are the TPR, EOI, and the ICR registers. The ICR low and ICR high registers are combined into one MSR. For performance reasons, the guest operating system should follow the hypervisor recommendation for the usage of the APIC MSRs.

MSR address	Register Name	Description
0x40000070	HV_X64_MSR_EOI	Accesses the APIC EOI
0x40000071	HV_X64_MSR_ICR	Accesses the APIC ICR-high and ICR-low
0x40000072	HV_X64_MSR_TPR	Access the APIC TPR

HV_X64_MSR_EOI

Bits	Description	Attributes
63:32	RsvdZ (reserved, should be zero)	Write
31:0	EOI value	Write

HV_X64_MSR_ICR

Bits	Description	Attributes
63:32	ICR high value	Read / Write
31:0	ICR low value	Read / Write

HV_X64_MSR_TPR

Bits	Description	Attributes
63:8	RsvdZ (reserved, should be zero)	Read / Write
7:0	TPR value	Read / Write

This MSR is intended to accelerate access to the TPR in 32-bit mode guest partitions. 64-bit mode guest partitions should set the TPR by way of CR8.

Synthetic Cluster IPI

A hypervisor supports hypercalls which allow to send virtual fixed interrupts to an arbitrary set of virtual processors.

Hypercall	Description
HvCallSendSyntheticClusterIpi	Sends a virtual fixed interrupt to the specified virtual processor set.
HvCallSendSyntheticClusterIpiEx	Similar to HvCallSendSyntheticClusterIpi, takes a sparse VP set as input.

EOI Assist

One field in the Virtual Processor Assist Page is the EOI Assist field. The EOI Assist field resides at offset 0 of the overlay page and is 32-bits in size. The format of the EOI assist field is as follows:

Bits	Description	Attributes
31:1	RsvdZ	Read / Write
0	No EOI Required	Read / Write

The guest OS performs an EOI by atomically writing zero to the EOI Assist field of the virtual VP assist page and checking whether the “No EOI required” field was previously

zero. If it was, the OS must write to the HV_X64_APIC_EOI MSR thereby triggering an intercept into the hypervisor. The following code is recommended to perform an EOI:

```
lea rcx, [VirtualApicAssistVa]
btr [rcx], 0
jc NoEoiRequired

mov ecx, HV_X64_APIC_EOI
wrmsr

NoEoiRequired:
```

The hypervisor sets the “No EOI required” bit when it injects a virtual interrupt if the following conditions are satisfied:

- The virtual interrupt is edge-triggered, and
- There are no lower priority interrupts pending

If, at a later time, a lower priority interrupt is requested, the hypervisor clears the “No EOI required” such that a subsequent EOI causes an intercept.

In case of nested interrupts, the EOI intercept is avoided only for the highest priority interrupt. This is necessary since no count is maintained for the number of EOIs performed by the OS. Therefore only the first EOI can be avoided and since the first EOI clears the “No EOI Required” bit, the next EOI generates an intercept. However nested interrupts are rare, so this is not a problem in the common case.

Note that devices and/or the I/O APIC (physical or synthetic) need not be notified of an EOI for an edge-triggered interrupt – the hypervisor intercepts such EOIs only to update the virtual APIC state. In some cases, the virtual APIC state can be lazily updated – in such cases, the “NoEoiRequired” bit is set by the hypervisor indicating to the guest that an EOI intercept is not necessary. At a later instant, the hypervisor can derive the state of the local APIC depending on the current value of the “NoEoiRequired” bit.

Enabling and disabling this enlightenment can be done at any time independently of the interrupt activity and the APIC state at that moment. While the enlightenment is enabled, conventional EOIs can still be performed irrespective of the “No EOI required” value but they will not realize the performance benefit of the enlightenment.

Inter-Partition Communication

Article • 05/26/2021

The hypervisor provides two simple mechanisms for one partition to communicate with another: messages and events. In both cases, notification is signaled by using the SynIC (synthetic interrupt controller).

SynIC Messages

The hypervisor provides a simple inter-partition communication facility that allows one partition to send a parameterized message to another partition. (Because the message is sent asynchronously, it is said to be posted.) The destination partition may be notified of the arrival of this message through an interrupt. Messages may be sent explicitly using the HvCallPostMessage hypercall or implicitly by the hypervisor.

Messages

When a message is sent, the hypervisor selects a free message buffer. The set of available message buffers depends on the event that triggered the sending of the message.

The hypervisor marks the message buffer "in use" and fills in the message header with the message type, payload size, and information about the sender. Finally, it fills in the message payload. The contents of the payload depend on the event that triggered the message.

The hypervisor then appends the message buffer to a receiving message queue. The receiving message queue depends on the event that triggered the sending of the message. For all message types, SINTx is either implicit (in the case of intercept messages), explicit (in the case of timer messages) or specified by a port ID (in the case of guest messages). The target virtual processor is either explicitly specified or chosen by the hypervisor when the message is enqueued. Virtual processors whose SynIC or SIM page is disabled will not be considered as potential targets. If no targets are available, the hypervisor terminates the operation and returns an error to the caller.

The hypervisor then determines whether the specified SINTx message slot within the SIM page for the target virtual processor is empty. If the message type in the message slot is equal to HvMessageTypeNone (that is, zero), the message slot is assumed to be empty. In this case, the hypervisor dequeues the message buffer and copies its contents to the message slot within the SIM page. The hypervisor may copy only the number of

payload bytes associated with the message. The hypervisor also attempts to generate an edge-triggered interrupt for the specified SINTx. If the APIC is software disabled or the SINTx is masked, the interrupt is lost. The arrival of this interrupt notifies the guest that a new message has arrived. If the SIM page is disabled or the message slot within the SIM page is not empty, the message remains queued, and no interrupt is generated.

As with any fixed-priority interrupt, the interrupt is not acknowledged by the virtual processor until the PPR (process priority register) is less than the vector specified in the SINTx register and interrupts are not masked by the virtual processor (rFLAGS[IF] is set to 1).

Multiple message buffers with the same SINTx can be queued to a virtual processor. In this case, the hypervisor will deliver the first message (that is, write it to the SIM page) and leave the others queued until one of three events occur:

- Another message buffer is queued.
- The guest indicates the “end of interrupt” by writing to the APIC’s EOI register.
- The guest indicates the “end of message” by writing to the SynIC’s EOM register.

In all three cases, the hypervisor will scan one or more message buffer queues and attempt to deliver additional messages. The hypervisor also attempts to generate an edge-triggered interrupt, indicating that a new message has arrived.

SIM Page

The SIM page consists of a 16-element array of 256-byte messages (see HV_MESSAGE data structure). Each array element (also known as a message slot) corresponds to a single synthetic interrupt source (SINTx). A message slot is said to be “empty” if the message type of the message in the slot is equal to HvMessageTypeNone.

The address for the SIM page is specified in the SIMP register. The address of the SIM page should be unique for each virtual processor. Programming these pages to overlap other instances of the SIEF or SIM pages or any other overlay page (for example, the hypercall page) will result in undefined behavior.

Read and write accesses by a virtual processor to the SIM page behave like read and write accesses to RAM. However, the hypervisor’s SynIC implementation also writes to the pages in response to certain events.

Upon virtual processor creation and reset, the SIM page is cleared to zero.

Recommended Message Handling

The SynIC message delivery mechanism is designed to accommodate efficient delivery and receipt of messages within a target partition. It is recommended that the message handling ISR (interrupt service routine) within the target partition perform the following steps:

- Examine the message that was deposited into the SIM message slot.
- Copy the contents of the message to another location and set the message type within the message slot to HvMessageTypeNone.
- Indicate the end of interrupt for the vector by writing to the APIC's EOI register.
- Perform any actions implied by the message.

Message Sources

The classes of events that can trigger the sending of a message are as follows:

- Intercepts: Any intercept in a virtual processor will cause a message to be sent to either the parent partition or a higher VTL.
- Timers: The timer mechanisms will cause messages to be sent. Associated with each virtual processor are four dedicated timer message buffers, one for each timer. The receiving message queue belongs to SINTx of the virtual processor whose timer triggered the sending of the message.
- Guest messages: The hypervisor supports message passing as an inter-partition communication mechanism between guests. The interfaces defined in this section allow one guest to send messages to another guest. The message buffers used for messages of this class are taken from the receiver's per-port pool of guest message buffers.

Message Buffers

A message buffer is used internally to the hypervisor to store a message until it is delivered to the recipient. The hypervisor maintains several sets of message buffers.

Guest Message Buffers

The hypervisor maintains a set of guest message buffers for each port. These buffers are used for messages sent explicitly from one partition to another by a guest. When a port is created, the hypervisor will allocate sixteen (16) message buffers from the port owner's memory pool. These message buffers are returned to the memory pool when the port is deleted.

Message Buffer Queues

For each partition and each virtual processor in the partition, the hypervisor maintains one queue of message buffers for each SINTx (synthetic interrupt source) in the virtual processor's SynIC. All message queues of a virtual processor are empty upon creation or reset of the virtual processor.

Reliability and Sequencing of Guest Message Buffers

Messages successfully posted by a guest have been queued for delivery by the hypervisor. Actual delivery and reception by the target partition is dependent upon its correct operation. Partitions may disable delivery of messages to particular virtual processors by either disabling its SynIC or disabling the SIMP.

Breaking a connection will not affect undelivered (queued) messages. Deletion of the target port will always free all of the port's message buffers, whether they are available or contain undelivered (queued) messages.

Messages arrive in the order in which they have been successfully posted. If the receiving port is associated with a specific virtual processor, then messages will arrive in the same order in which they were posted. If the receiving port is associated with HV_ANY_VP, then messages are not guaranteed to arrive in any particular order.

SynIC Event Flags

In addition to messages, the SynIC supports a second type of cross-partition notification mechanism called event flags. Event flags may be set explicitly using the HvCallSignalEvent hypercall or implicitly by the hypervisor.

Event Flags versus Messages

Event flags are lighter-weight than messages and are therefore lower overhead. Furthermore, event flags do not require any buffer allocation or queuing within the hypervisor, so HvCallSignalEvent will never fail due to insufficient resources.

Event Flag Delivery

When a partition calls HvCallSignalEvent, it specifies an event flag number. The hypervisor responds by atomically setting a bit within the receiving virtual processor's SIEF page. Virtual processors whose SynIC or SIEF page is disabled will not be considered as potential targets. If no targets are available, the hypervisor terminates the operation and returns an error to the caller.

If the event flag was previously cleared, the hypervisor attempts to notify the receiving partition that the flag is now set by generating an edge-triggered interrupt. The target virtual processor, along with the target SINTx, is specified as part of a port's creation. If the SINTx is masked, HvSignalEvent returns HV_STATUS_INVALID_SYNIC_STATE.

As with any fixed-priority external interrupt, the interrupt is not acknowledged by the virtual processor until the process priority register (PPR) is less than the vector specified in the SINTx register and interrupts are not masked by the virtual processor (rFLAGS[IF] is set to 1).

SIEF Page

The SIEF page consists of a 16-element array of 256-byte event flags (see HV_SYNIC_EVENT_FLAGS). Each array element corresponds to a single synthetic interrupt source (SINTx).

The address for the SIEF page is specified in the SIEF register. The address of the SIEF page should be unique for each virtual processor. Programming these pages to overlap other instances of the SIEF or SIM pages or any other overlay page (for example, the hypercall page) will result in undefined behavior.

Read and write accesses by a virtual processor to the SIEF page behave like read and write accesses to RAM. However, the hypervisor's SynIC implementation also writes to the pages in response to certain events.

Upon virtual processor creation and reset, the SIEF page is cleared to zero.

Recommended Event Flag Handling

It is recommended that the event flag interrupt service routine (ISR) within the target partition perform the following steps:

- Examine the event flags and determine which ones, if any, are set.
- Clear one or more event flags by using a locked (atomic) operation such as LOCK AND or LOCK CMPXCHG.
- Indicate the end of interrupt for the vector by writing to the APIC's EOI register.
- Perform any actions implied by the event flags that were set.

Ports and Connections

A message or event sent from one guest to another must be sent through a pre-allocated connection. A connection, in turn, must be associated with a destination port.

A port is allocated from the receiver's memory pool and specifies which virtual processor and SINTx to target. Event ports have a "base flag number" and "flag count" that allow the caller to specify a range of valid event flags for that port.

Connections are allocated from the sender's memory pool. When a connection is created, it must be associated with a valid port. This binding creates a simple, one-way communication channel. If a port is subsequently deleted, its connection, while it remains, becomes useless.

SynIC MSRs

In addition to the memory-mapped registers defined for a local APIC, the following model-specific registers (MSRs) are defined in the SynIC. Each virtual processor has its own copy of these registers, so they can be programmed independently.

MSR Address	Register Name	Function
0x40000080	SCONTROL	SynIC Control
0x40000081	SVERSION	SynIC Version
0x40000082	SIEFP	Interrupt Event Flags Page
0x40000083	SIMP	Interrupt Message Page
0x40000084	EOM	End of message
0x40000090	SINT0	Interrupt source 0 (hypervisor)
0x40000090	SINT1	Interrupt source 1
0x40000090	SINT2	Interrupt source 2
0x40000090	SINT3	Interrupt source 3
0x40000090	SINT4	Interrupt source 4
0x40000090	SINT5	Interrupt source 5
0x40000090	SINT6	Interrupt source 6
0x40000090	SINT7	Interrupt source 7
0x40000090	SINT8	Interrupt source 8
0x40000090	SINT9	Interrupt source 9
0x40000090	SINT10	Interrupt source 10

MSR Address	Register Name	Function
0x40000090	SINT11	Interrupt source 11
0x40000090	SINT12	Interrupt source 12
0x40000090	SINT13	Interrupt source 13
0x40000090	SINT14	Interrupt source 14
0x40000090	SINT15	Interrupt source 15

SCONTROL Register

This register is used to control SynIC behavior of the virtual processor.

At virtual processor creation time and upon processor reset, the value of this SCONTROL (SynIC control register) is 0x0000000000000000. Thus, message queuing and event flag notifications will be disabled.

Bits	Field	Description	Attributes
63:1	RsvdP	Value must be preserved	Read / write
0	Enable	When set, this virtual processor will allow message queuing and event flag notifications to be posted to its SynIC. When clear, message queuing and event flag notifications cannot be directed to this virtual processor.	Read / write

SVERSION Register

This is a read-only register, and it returns the version number of the SynIC. Attempts to write to this register result in a #GP fault.

Bits	Field	Description	Attributes
63:32	RsvdP		Read
31:0	SynIC Version	Version number of the SynIc	Read

SIEFP Register

At virtual processor creation time and upon processor reset, the value of this SIEFP (synthetic interrupt event flags page) register is 0x0000000000000000. Thus, the SIEFP is

disabled by default. The guest must enable it by setting bit 0. If the specified base address is beyond the end of the partition's GPA space, the SIEFP page will not be accessible to the guest. When modifying the register, guests should preserve the value of the reserved bits (1 through 11) for future compatibility.

Bits	Field	Description	Attributes
63:12	Base Address	Base address (in GPA space) of SIEFP (low 12 bits assumed to be disabled)	Read / Write
11:1	RsvdP	Reserved, value should be preserved	Read / Write
0	Enable	SIEFP enable	Read / Write

SIMP Register

At virtual processor creation time and upon processor reset, the value of this SIMP (synthetic interrupt message page) register is 0x0000000000000000. Thus, the SIMP is disabled by default. The guest must enable it by setting bit 0. If the specified base address is beyond the end of the partition's GPA space, the SIMP page will not be accessible to the guest. When modifying the register, guests should preserve the value of the reserved bits (1 through 11) for future compatibility.

Bits	Field	Description	Attributes
63:12	Base Address	Base address (in GPA space) of SIMP (low 12 bits assumed to be disabled)	Read / Write
11:1	RsvdP	Reserved, value should be preserved	Read / Write
0	Enable	SIMP enable	Read / Write

SINTx Registers

At virtual processor creation time, the default value of all SINTx (synthetic interrupt source) registers is 0x0000000000010000. Thus, all synthetic interrupt sources are masked by default. The guest must unmask them by programming an appropriate vector and clearing bit 16.

Setting the polling bit will have the effect of unmasking an interrupt source, except that an actual interrupt is not generated.

The AutoEOI flag indicates that an implicit EOI should be performed by the hypervisor when an interrupt is delivered to the virtual processor. In addition, the hypervisor will automatically clear the corresponding flag in the “in-service register” (ISR) of the virtual APIC. If the guest enables this behavior, then it must not perform an EOI in its interrupt service routine. The AutoEOI flag can be turned on at any time, though the guest must perform an explicit EOI on an in-flight interrupt The timing consideration makes it difficult to know whether a particular interrupt needs EOI or not, so it is recommended that once SINT is unmasked, its settings are not changed. Likewise, the AutoEOI flag can be turned off at any time, though the same concerns about in-flight interrupts apply

Valid values for vector are 16-255 inclusive. Specifying an invalid vector number results in #GP.

Bits	Field	Description	Attributes
63:19	RsvdP	Reserved, value should be preserved	Read / Write
18	Polling	Enables polling mode	Read / Write
17	AutoEOI	Set if an implicit EOI should be performed upon interrupt delivery	Read / Write
16	Masked	Set if the SINT is masked	Read / Write
15:8	RsvdP	Reserved, value should be preserved	Read / Write
7:0	Vector	Interrupt vector	Read / Write

EOM Register

A write to the end of message (EOM) register by the guest causes the hypervisor to scan the internal message buffer queue(s) associated with the virtual processor. If a message buffer queue contains a queued message buffer, the hypervisor attempts to deliver the message. Message delivery succeeds if the SIM page is enabled and the message slot corresponding to the SINTx is empty (that is, the message type in the header is set to HvMessageTypeNone). If a message is successfully delivered, its corresponding internal message buffer is dequeued and marked free. If the corresponding SINTx is not masked, an edge-triggered interrupt is delivered (that is, the corresponding bit in the IRR is set).

This register can be used by guests to “poll” for messages. It can also be used as a way to drain the message queue for a SINTx that has been disabled (that is, masked).

If the message queues are all empty, a write to the EOM register is a no-op.

Reads from the EOM register always returns zeros.

Bits	Field	Description	Attributes
63:0	RsvdZ	Write-only trigger	Write

Timers

Article • 05/26/2021

The hypervisor provides simple timing services. These are based on a constant-rate reference time source (typically the ACPI timer on x64 systems).

The following timer services are provided:

- A per-partition reference time counter.
- Four synthetic timers per virtual processor. Each synthetic timer is a single-shot or periodic timer that delivers a message or asserts an interrupt when it expires.
- One virtual APIC timer per virtual processor.
- A partition reference time enlightenment, based on the host platform's support for an Invariant Time Stamp Counter (iTSC).

Reference Counter

The hypervisor maintains a per-partition reference time counter. It has the characteristic that successive accesses to it return strictly monotonically increasing (time) values as seen by any and all virtual processors of a partition. Furthermore, the reference counter is rate constant and unaffected by processor or bus speed transitions or deep processor power savings states. A partition's reference time counter is initialized to zero when the partition is created. The reference counter for all partitions count at the same rate, but at any time, their absolute values will typically differ because partitions will have different creation times.

The reference counter continues to count up as long as at least one virtual processor is not explicitly suspended.

Partition Reference Counter MSR

A partition's reference counter can be accessed through a partition-wide MSR.

MSR address	Register Name	Description
0x40000020	HV_X64_MSR_TIME_REF_COUNT	Time reference count (partition-wide)

When a partition is created, the value of the TIME_REF_COUNT MSR is set to 0x0000000000000000. This value cannot be modified by a virtual processor. Any attempt to write to it results in a #GP fault.

Partition Reference Time Enlightenment

The partition reference time enlightenment presents a reference time source to a partition which does not require an intercept into the hypervisor. This enlightenment is available only when the underlying platform provides support of an invariant processor Time Stamp Counter (TSC), or iTSC. In such platforms, the processor TSC frequency remains constant irrespective of changes in the processor's clock frequency due to the use of power management states such as ACPI processor performance states, processor idle sleep states (ACPI C-states), etc.

The partition reference time enlightenment uses a virtual TSC value, an offset and a multiplier to enable a guest partition to compute the normalized reference time since partition creation, in 100nS units. The mechanism also allows a guest partition to atomically compute the reference time when the guest partition is migrated to a platform with a different TSC rate, and provides a fallback mechanism to support migration to platforms without the constant rate TSC feature.

This facility is not intended to be used a source of wall clock time, since the reference time computed using this facility will appear to stop during the time that a guest partition is saved until the subsequent restore.

Partition Reference Time Stamp Counter Page

The hypervisor provides a partition-wide virtual reference TSC page which is overlaid on the partition's GPA space. A partition's reference time stamp counter page is accessed through the Reference TSC MSR.

The reference TSC page is defined using the following structure:

```c
typedef struct
{
   volatile UINT32 TscSequence;
   UINT32 Reserved1;
   volatile UINT64 TscScale;
   volatile INT64 TscOffset;
   UINT64 Reserved2[509];
} HV_REFERENCE_TSC_PAGE;
```

Reference Time Stamp Counter (TSC) Page MSR

A guest wishing to access its reference TSC page must use the following model-specific register (MSR). A partition which possesses the AccessPartitionReferenceTsc privilege may access the MSR.

MSR address	Register Name	Description
0x40000021	HV_X64_MSR_REFERENCE_TSC	Reference TSC page

Bits	Description	Attributes
63:12	GPA Page Number	Read / Write
11:1	RsvdP (value should be preserved)	Read / Write
0	Enable	Read / Write

At the guest partition creation time, the value of the reference TSC MSR is 0x0000000000000000. Thus, the reference TSC page is disabled by default. The guest must enable the reference TSC page by setting bit 0. If the specified base address is beyond the end of the partition's GPA space, the reference TSC page will not be accessible to the guest. When modifying the register, guests should preserve the value of the reserved bits (1 through 11) for future compatibility.

Partition Reference TSC Mechanism

The partition reference time is computed by the following formula:

ReferenceTime = ((VirtualTsc * TscScale) >> 64) + TscOffset

The multiplication is a 64 bit multiplication, which results in a 128 bit number which is then shifted 64 times to the right to obtain the high 64 bits.

The TscScale value is used to adjust the Virtual TSC value across migration events to mitigate TSC frequency changes from one platform to another.

The TscSequence value is used to synchronize access to the enlightened reference time if the scale and/or the offset fields are changed during save/restore or live migration. This field serves as a sequence number which is incremented whenever the scale and/or the offset fields are modified. A special value of 0x0 is used to indicate that this facility is no longer a reliable source of reference time and the VM must fall back to a different source.

The recommended code for computing the partition reference time using this enlightenment is shown below:

```C
do
{
    StartSequence = ReferenceTscPage->TscSequence;
    if (StartSequence == 0)
    {
        // 0 means that the Reference TSC enlightenment is not available at
        // the moment, and the Reference Time can only be obtained from
        // reading the Reference Counter MSR.
        ReferenceTime = rdmsr(HV_X64_MSR_TIME_REF_COUNT);
        return ReferenceTime;
    }

    Tsc = rdtsc();

    // Assigning Scale and Offset should neither happen before
    // setting StartSequence, nor after setting EndSequence.
    Scale = ReferenceTscPage->TscScale;
    Offset = ReferenceTscPage->TscOffset;

    EndSequence = ReferenceTscPage->TscSequence;
} while (EndSequence != StartSequence);

// The result of the multiplication is treated as a 128-bit value.
ReferenceTime = ((Tsc * Scale) >> 64) + Offset;
return ReferenceTime;
```

Synthetic Timers

Synthetic timers provide a mechanism for generating an interrupt after some specified time in the future. Both one-shot and periodic timers are supported. A synthetic timer sends a message to a specified SynIC SINTx (synthetic interrupt source) upon expiration, or asserts an interrupt, depending on how it is configured.

The hypervisor guarantees that a timer expiration signal will never be delivered before the expiration time. The signal may arrive any time after the expiration time.

Periodic Timers

The hypervisor attempts to signal periodic timers on a regular basis. However, if the virtual processor used to signal the expiration is not available, some of the timer expirations may be delayed. A virtual processor may be unavailable because it is suspended (for example, during intercept handling) or because the hypervisor's scheduler decided that the virtual processor should not be scheduled on a logical

processor (for example, because another virtual processor is using the logical processor or the virtual processor has exceeded its quota).

If a virtual processor is unavailable for a sufficiently long period of time, a full timer period may be missed. In this case, the hypervisor uses one of two techniques.

The first technique involves timer period modulation, in effect shortening the period until the timer “catches up”. If a significant number of timer signals have been missed, the hypervisor may be unable to compensate by using period modulation. In this case, some timer expiration signals may be skipped completely.

For timers that are marked as lazy, the hypervisor uses a second technique for dealing with the situation in which a virtual processor is unavailable for a long period of time. In this case, the timer signal is deferred until this virtual processor is available. If it doesn’t become available until shortly before the next timer is due to expire, it is skipped entirely.

Ordering of Timer Expirations

Synthetic and virtualized timers generate interrupts at or near their designated expiration time. Due to hardware and other scheduling interactions, interrupts could potentially be delayed. No ordering may be assumed between any set of timers.

Direct Synthetic Timers

“Direct” synthetic timers assert an interrupt upon timer expiration instead of sending a message to a SynIc synthetic interrupt source. A synthetic timer is set to “direct” mode by setting the “DirectMode” field of the synthetic timer configuration MSRs. The “ApicVector” field controls the interrupt vector that is asserted upon timer expiration.

Synthetic Timer MSRs

Synthetic timers are configured by using model-specific registers (MSRs) associated with each virtual processor. Each of the four synthetic timers has an associated pair of MSRs.

MSR address	Register Name	Description
0x400000B0	HV_X64_MSR_STIMER0_CONFIG	Configuration register for synthetic timer 0.
0x400000B1	HV_X64_MSR_STIMER0_COUNT	Expiration time or period for synthetic timer 0.
0x400000B2	HV_X64_MSR_STIMER1_CONFIG	Configuration register for synthetic timer 1.

MSR address	Register Name	Description
0x400000B3	HV_X64_MSR_STIMER1_COUNT	Expiration time or period for synthetic timer 1.
0x400000B4	HV_X64_MSR_STIMER2_CONFIG	Configuration register for synthetic timer 2.
0x400000B5	HV_X64_MSR_STIMER2_COUNT	Expiration time or period for synthetic timer 2.
0x400000B6	HV_X64_MSR_STIMER3_CONFIG	Configuration register for synthetic timer 3.
0x400000B7	HV_X64_MSR_STIMER4_COUNT	Expiration time or period for synthetic timer 3.

Synthetic Timer Configuration Register

Bits	Description	Attributes
63:20	RsvdZ (value should be set to zero)	Read / Write
19:16	SINTx - synthetic interrupt source	Read / Write
15:13	RsvdZ (value should be set to zero)	Read / Write
12	Direct Mode - Assert and interrupt upon timer expiration.	Read / Write
11:4	ApicVector - Controls the asserted interrupt vector in direct mode	Read / Write
3	AutoEnable - Set if writing the corresponding counter implicitly causes the timer to be enabled	Read / Write
2	Lazy - Set if timer is lazy	Read / Write
1	Periodic - Set if timer is periodic	Read / Write
0	Enabled - set if timer is enabled	Read / Write

When a virtual processor is created and reset, the value of all HV_X64_MSR_STIMERx_CONFIG (synthetic timer configuration) registers is set to 0x0000000000000000. Thus, all synthetic timers are disabled by default.

If AutoEnable is set, then writing a non-zero value to the corresponding count register will cause Enable to be set and activate the counter. Otherwise, Enable should be set

after writing the corresponding count register in order to activate the counter. For information about the Count register, see the following section.

When a one-shot timer expires, it is automatically marked as disabled. Periodic timers remain enabled until explicitly disabled.

If a one-shot is enabled and the specified count is in the past, it will expire immediately.

It is not permitted to set the SINTx field to zero for an enabled timer (that is not in direct mode). If attempted, the timer will be marked disabled (that is, bit 0 cleared) immediately.

Writing the configuration register of a timer that is already enabled may result in undefined behavior. For example, merely changing a timer from one-shot to periodic may not produce what is intended. Timers should always be disabled prior to changing any other properties.

Synthetic Timer Count Register

Bits	Description	Attributes
63:0	Count—expiration time for one-shot timers, duration for periodic timers	Read / Write

The value programmed into the Count register is a time value measured in 100 nanosecond units. Writing the value zero to the Count register will stop the counter, thereby disabling the timer, independent of the setting of AutoEnable in the configuration register.

Note that the Count register is permitted to wrap. Wrapping will have no effect on the behavior of the timer, regardless of any timer property.

For one-shot timers, it represents the absolute timer expiration time. The timer expires when the reference counter for the partition is equal to or greater than the specified count value.

For periodic timers, the count represents the period of the timer. The first period begins when the synthetic timer is enabled.

Synthetic Timer Expiration Message

Timer expiration messages are sent when a timer event fires. Refer to the HV_TIMER_MESSAGE_PAYLOAD for the definition of the message payload.

Synthetic Time-Unhalted Timer MSRs

Synthetic Time-Unhalted Timer MSRs are available if a partition has the AccessSyntheticTimerRegs privilege and EDX bit 23 in the Hypervisor Feature Identification CPUID leaf 0x40000003 is set. Guest software may program the synthetic time-unhalted timer to generate a periodic interrupt after executing for a specified amount of time in 100ns units. When the interrupt fires, the SyntheticTimeUnhaltedTimerExpired field in the VP Assist Page will be set to TRUE. Guest software may reset this field to FALSE. Unlike architectural performance counters, the synthetic timer is never reset by the hypervisor and runs continuously between interrupts. Vectors ==2 send an NMI, other vectors send a fixed interrupt.

Unlike regular synthetic timers that accumulate time when the guest has halted (ie: gone idle), the Synthetic Time-Unhalted Timer accumulates time only while the guest is not halted.

MSR address	Register Name	Description
0x40000114	HV_X64_MSR_STIME_UNHALTED_TIMER_CONFIG	Synthetic Time-Unhalted Timer Configuration MSR
0x40000115	HV_X64_MSR_STIME_UNHALTED_TIMER_COUNT	Synthetic Time-Unhalted Timer Count MSR

Synthetic Time-Unhalted Timer Configuration MSR

Bits	Description	Attributes
63:9	RsvdZ (value should be set to zero)	Read / Write
8	Enabled	Read / Write
7:0	Vector	Read / Write

Synthetic Time-Unhalted Timer Count MSR

Bits	Description	Attributes
63:0	Periodic rate of interrupts in 100 ns units	Read / Write

Virtual Secure Mode

Article • 07/07/2022

Virtual Secure Mode (VSM) is a set of hypervisor capabilities and enlightenments offered to host and guest partitions which enables the creation and management of new security boundaries within operating system software. VSM is the hypervisor facility on which Windows security features including Device Guard, Credential Guard, virtual TPMs and shielded VMs are based. These security features were introduced in Windows 10 and Windows Server 2016.

VSM enables operating system software in the root and guest partitions to create isolated regions of memory for storage and processing of system security assets. Access to these isolated regions is controlled and granted solely through the hypervisor, which is a highly privileged, highly trusted part of the system's Trusted Compute Base (TCB). Because the hypervisor runs at a higher privilege level than operating system software and has exclusive control of key system hardware resources such as memory access permission controls in the CPU MMU and IOMMU early in system initialization, the hypervisor can protect these isolated regions from unauthorized access, even from operating system software (e.g., OS kernel and device drivers) with supervisor mode access (i.e. CPL0, or "Ring 0").

With this architecture, even if normal system level software running in supervisor mode (e.g. kernel, drivers, etc.) is compromised by malicious software, the assets in isolated regions protected by the hypervisor can remain secured.

Virtual Trust Level (VTL)

VSM achieves and maintains isolation through Virtual Trust Levels (VTLs). VTLs are enabled and managed on both a per-partition and per-virtual processor basis.

Virtual Trust Levels are hierarchical, with higher levels being more privileged than lower levels. VTL0 is the least privileged level, with VTL1 being more privileged than VTL0, VTL2 being more privileged than VTL1, etc.

Architecturally, up to 16 levels of VTLs are supported; however a hypervisor may choose to implement fewer than 16 VTL's. Currently, only two VTLs are implemented.

```c
typedef UINT8 HV_VTL, *PHV_VTL;

#define HV_NUM_VTLS 2
```

```
#define HV_INVALID_VTL ((HV_VTL) -1)
#define HV_VTL_ALL 0xF
```

Each VTL has its own set of memory access protections. These access protections are managed by the hypervisor in a partition's physical address space, and thus cannot be modified by system level software running in the partition.

Since more privileged VTLs can enforce their own memory protections, higher VTLs can effectively protect areas of memory from lower VTLs. In practice, this allows a lower VTL to protect isolated memory regions by securing them with a higher VTL. For example, VTL0 could store a secret in VTL1, at which point only VTL1 could access it. Even if VTL0 is compromised, the secret would be safe.

VTL Protections

There are multiple facets to achieving isolation between VTLs:

- Memory Access Protections: Each VTL maintains a set of guest physical memory access protections. Software running at a particular VTL can only access memory in accordance with these protections.
- Virtual Processor State: Virtual processors maintain separate per-VTL state. For example, each VTL defines a set of a private VP registers. Software running at a lower VTL cannot access the higher VTL's private virtual processor's register state.
- Interrupts: Along with a separate processor state, each VTL also has its own interrupt subsystem (local APIC). This allows higher VTLs to process interrupts without risking interference from a lower VTL.
- Overlay Pages: Certain overlay pages are maintained per-VTL such that higher VTLs have reliable access. E.g. there is a separate hypercall overlay page per VTL.

VSM Detection and Status

The VSM capability is advertised to partitions via the AccessVsm partition privilege flag. Only partitions with all of the following privileges may utilize VSM: AccessVsm, AccessVpRegisters, and AccessSynicRegs.

VSM Capability Detection

Guests should use the following model-specific register to access a report on VSM capabilities:

MSR address	Register Name	Description

MSR address	Register Name	Description
0x000D0006	HV_X64_REGISTER_VSM_CAPABILITIES	Report on VSM capabilities.

The format of the Register VSM Capabilities MSR is as follows:

Bits	Description	Attributes
63	Dr6Shared	Read
62:47	MbecVtlMask	Read
46	DenyLowerVtlStartup	Read
45:0	RsvdZ	Read

Dr6Shared indicates to the guest whether Dr6 is a shared register between the VTLs.

MvecVtlMask indicates to the guest the VTLs for which Mbec can be enabled.

DenyLowerVtlStartup indicates to the guest whether a Vtl can deny a VP reset by a lower VTL.

VSM Status Register

In addition to a partition privilege flag, two virtual registers can be used to learn additional information about VSM status: `HvRegisterVsmPartitionStatus` and `HvRegisterVsmVpStatus`.

HvRegisterVsmPartitionStatus

HvRegisterVsmPartitionStatus is a per-partition read-only register that is shared across all VTLs. This register provides information about which VTLs have been enabled for the partition, which VTLs have Mode Based Execution Controls enabled, as well as the maximum VTL allowed.

C

```c
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 EnabledVtlSet : 16;
        UINT64 MaximumVtl : 4;
        UINT64 MbecEnabledVtlSet: 16;
```

```
        UINT64 ReservedZ : 28;
    };
} HV_REGISTER_VSM_PARTITION_STATUS;
```

HvRegisterVsmVpStatus

HvRegisterVsmVpStatus is a read-only register and is shared across all VTLs. It is a per-VP register, meaning each virtual processor maintains its own instance. This register provides information about which VTLs have been enabled, which is active, as well as the MBEC mode active on a VP.

```C
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 ActiveVtl : 4;
        UINT64 ActiveMbecEnabled : 1;
        UINT64 ReservedZ0 : 11;
        UINT64 EnabledVtlSet : 16;
        UINT64 ReservedZ1 : 32;
    };
} HV_REGISTER_VSM_VP_STATUS;
```

ActiveVtl is the ID of the VTL context that is currently active on the virtual processor.

ActiveMbecEnabled specifies that MBEC is currently active on the virtual processor.

EnabledVtlSet is a bitmap of the VTL's that are enabled on the virtual processor.

Partition VTL Initial state

When a partition starts or resets, it begins running in VTL0. All other VTLs are disabled at partition creation.

VTL Enablement

To begin using a VTL, a lower VTL must initiate the following:

1. Enable the target VTL for the partition. This makes the VTL generally available for the partition.

2. Enable the target VTL on one or more virtual processors. This makes the VTL available for a VP, and sets its initial context. It is recommended that all VPs have the same enabled VTLs. Having a VTL enabled on some VPs (but not all) can lead to unexpected behavior.
3. Once the VTL is enabled for a partition and VP, it can begin setting access protections once the EnableVtlProtection flag has been set.

Note that VTLs need not be consecutive.

Enabling a Target VTL for a Partition

The HvCallEnablePartitionVtl hypercall is used to enable a VTL for a certain partition. Note that before software can actually execute in a particular VTL, that VTL must be enabled on virtual processors in the partition.

Enabling a Target VTL for Virtual Processors

Once a VTL is enabled for a partition, it can be enabled on the partition's virtual processors. The HvCallEnableVpVtl hypercall can be used to enable VTLs for a virtual processor, which sets its initial context.

Virtual processors have one "context" per VTL. If a VTL is switched, the VTL's private state is also switched.

VTL Configuration

Once a VTL has been enabled, its configuration can be changed by a VP running at an equal or higher VTL.

Partition Configuration

Partition-wide attributes can be configured using the HvRegisterVsmPartitionConfig register. There is one instance of this register for each VTL (greater than 0) on every partition.

Every VTL can modify its own instance of HV_REGISTER_VSM_PARTITION_CONFIG, as well as instances for lower VTLs. VTLs may not modify this register for higher VTLs.

C

```c
typedef union
{
```

```
        UINT64 AsUINT64;
        struct
        {
            UINT64 EnableVtlProtection : 1;
            UINT64 DefaultVtlProtectionMask : 4;
            UINT64 ZeroMemoryOnReset : 1;
            UINT64 DenyLowerVtlStartup : 1;
            UINT64 ReservedZ : 2;
            UINT64 InterceptVpStartup : 1;
            UINT64 ReservedZ : 54; };
    } HV_REGISTER_VSM_PARTITION_CONFIG;
```

The fields of this register are described below.

Enable VTL Protections

Once a VTL has been enabled, the EnableVtlProtection flag must be set before it can begin applying memory protections. This flag is write-once, meaning that once it has been set, it cannot be modified.

Default Protection Mask

By default, the system applies RWX protections to all currently mapped pages, and any future “hot-added” pages. Hot-added pages refer to any memory that is added to a partition during a resize operation.

A higher VTL can set a different default memory protection policy by specifying DefaultVtlProtectionMask in HV_REGISTER_VSM_PARTITION_CONFIG. This mask must be set at the time the VTL is enabled. It cannot be changed once it is set, and is only cleared by a partition reset.

Bit	Description
0	Read
1	Write
2	Kernel Mode Execute (KMX)
3	User Mode Execute (UMX)

Zero Memory on Reset

ZeroMemOnReset is a bit that controls if memory is zeroed before a partition is reset. This configuration is on by default. If the bit is set, the partition’s memory is zeroed

upon reset so that a higher VTL's memory cannot be compromised by a lower VTL. If this bit is cleared, the partition's memory is not zeroed on reset.

DenyLowerVtlStartup

The DenyLowerVtlStartup flag controls if a virtual processor may be started or reset by lower VTLs. This includes architectural ways of resetting a virtual processor (e.g. SIPI on X64) as well as the HvCallStartVirtualProcessor hypercall.

InterceptVpStartup

If InterceptVpStartup flag is set, starting or resetting a virtual processor generates an intercept to the higher VTL.

Configuring Lower VTLs

The following register can be used by higher VTLs to configure the behavior of lower VTLs:

C

```c
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 MbecEnabled : 1;
        UINT64 TlbLocked : 1;
        UINT64 ReservedZ : 62;
    };
} HV_REGISTER_VSM_VP_SECURE_VTL_CONFIG;
```

Each VTL (higher than 0) has an instance of this register for every VTL lower than itself. For example, VTL2 would have two instances of this register – one for VTL1, and a second for VTL0.

The fields of this register are described below.

MbecEnabled

This field configures whether MBEC is enabled for the lower VTL.

TlbLocked

This field locks the lower VTL's TLB. This capability can be used to prevent lower VTLs from causing TLB invalidations which might interfere with a higher VTL. When this bit is set, all address space flush requests from the lower VTL are blocked until the lock is lifted.

To unlock the TLB, the higher VTL can clear this bit. Also, once a VP returns to a lower VTL, it releases all TLB locks which it holds at the time.

VTL Entry

A VTL is "entered" when a VP switches from a lower VTL to a higher one. This can happen for the following reasons:

1. VTL call: this is when software explicitly wishes to invoke code in a higher VTL.
2. Secure interrupt: if an interrupt is received for a higher VTL, the VP will enter the higher VTL.
3. Secure intercept: certain actions will trigger a secure interrupt (accessing certain MSRs for example).

Once a VTL is entered, it must voluntarily exit. A higher VTL cannot be preempted by a lower VTL.

Identifying VTL Entry Reason

In order to react appropriately to an entry, a higher VTL might need to know the reason it was entered. To discern between entry reasons, the VTL entry is included in the HV_VP_VTL_CONTROL structure.

VTL Call

A "VTL call" is when a lower VTL initiates an entry into a higher VTL (for example, to protect a region of memory with the higher VTL) through the HvCallVtlCall hypercall.

VTL calls preserve the state of shared registers across VTL switches. Private registers are preserved on a per-VTL level. The exception to these restrictions are the registers required by the VTL call sequence. The following registers are required for a VTL call:

x64	x86	Description
RCX	EDX:EAX	Specifies a VTL call control input to the hypervisor
RAX	ECX	Reserved

All bits in the VTL call control input are currently reserved.

VTL Call Restrictions

VTL calls can only be initiated from the most privileged processor mode. For example, on x64 systems a VTL call can only come from CPL0. A VTL call initiated from a processor mode which is anything but the most privileged on the system results in the hypervisor injecting a #UD exception into the virtual processor.

A VTL call can only switch into the next highest VTL. In other words, if there are multiple VTLs enabled, a call cannot “skip” a VTL. The following actions result in a #UD exception:

- A VTL call initiated from a processor mode which is anything but the most privileged on the system (architecture specific).
- A VTL call from real mode (x86/x64)
- A VTL call on a virtual processor where the target VTL is disabled (or has not been already enabled).
- A VTL call with an invalid control input value

VTL Exit

A switch to a lower VTL is known as a “return”. Once a VTL has finished processing, it can initiate a VTL return in order to switch to a lower VTL. The only way a VTL return can occur is if a higher VTL voluntarily initiates one. A lower VTL can never preempt a higher one.

VTL Return

A “VTL return” is when a higher VTL initiates a switch into a lower VTL through the HvCallVtlReturn hypercall. Similar to a VTL call, private processor state is switched out, and shared state remains in place. If the lower VTL has explicitly called into the higher VTL, the hypervisor increments the higher VTL’s instruction pointer before the return is complete so that it may continue after a VTL call.

A VTL Return code sequence requires the use of the following registers:

x64	x86	Description
RCX	EDX:EAX	Specifies a VTL return control input to the hypervisor
RAX	ECX	Reserved

The VTL return control input has the following format:

Bits	Field	Description
63:1	RsvdZ	
0	Fast return	Registers are not restored

The following actions will generate a #UD exception:

- Attempting a VTL return when the lowest VTL is currently active
- Attempting a VTL return with an invalid control input value
- Attempting a VTL return from a processor mode which is anything but the most privileged on the system (architecture specific)

Fast Return

As a part of processing a return, the hypervisor can restore the lower VTL's register state from the HV_VP_VTL_CONTROL structure. For example, after processing a secure interrupt, a higher VTL may wish to return without disrupting the lower VTL's state. Therefore, the hypervisor provides a mechanism to simply restore the lower VTL's registers to their pre-call value stored in the VTL control structure.

If this behavior is not necessary, a higher VTL can use a "fast return". A fast return is when the hypervisor does not restore register state from the control structure. This should be utilized whenever possible to avoid unnecessary processing.

This field can be set with bit 0 of the VTL return input. If it is set to 0, the registers are restored from the HV_VP_VTL_CONTROL structure. If this bit is set to 1, the registers are not restored (a fast return).

Hypercall Page Assist

The hypervisor provides mechanisms to assist with VTL calls and returns via the hypercall page. This page abstracts the specific code sequence required to switch VTLs.

The code sequences to execute VTL calls and returns may be accessed by executing specific instructions in the hypercall page. The call/return chunks are located at an offset in the hypercall page determined by the HvRegisterVsmCodePageOffset virtual register. This is a read-only and partition-wide register, with a separate instance per-VTL.

A VTL can execute a VTL call/return using the CALL instruction. A CALL to the correct location in the hypercall page will initiate a VTL call/return.

```C
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 VtlCallOffset : 12;
        UINT64 VtlReturnOffset : 12;
        UINT64 ReservedZ : 40;
    };
} HV_REGISTER_VSM_CODE_PAGE_OFFSETS;
```

To summarize, the steps for calling a code sequence using the hypercall page are as follows:

1. Map the hypercall page into a VTL's GPA space
2. Determine the correct offset for the code sequence (VTL call or return).
3. Execute the code sequence using CALL.

Memory Access Protections

One necessary protection provided by VSM is the ability to isolate memory accesses.

Higher VTLs have a high degree of control over the type of memory access permissible by lower VTLs. There are three basic types of protections that can be specified by a higher VTL for a particular GPA page: Read, Write, and eXecute. These are defined in the following table:

Name	Description
Read	Controls whether read access is allowed to a memory page
Write	Controls whether write access allowed to a memory page
Execute	Controls whether instruction fetches are allowed for a memory page.

These three combine for the following types of memory protection:

1. No access
2. Read-only, no execute
3. Read-only, execute
4. Read/write, no execute
5. Read/write, execute

If “mode based execution control (MBEC)” is enabled, user and kernel mode execute protections can be set separately.

Higher VTLs can set the memory protection for a GPA through the HvCallModifyVtlProtectionMask hypercall.

Memory Protection Hierarchy

Memory access permissions can be set by a number of sources for a particular VTL. Each VTL’s permissions can potentially be restricted by a number of other VTLs, as well as by the host partition. The order in which protections are applied is the following:

1. Memory protections set by the host
2. Memory protections set by higher VTLs

In other words, VTL protections supersede host protections. Higher-level VTLs supersede lower-level VTLs. Note that a VTL may not set memory access permissions for itself.

A conformant interface is expected to not overlay any non-RAM type over RAM.

Memory Access Violations

If a VP running at a lower VTL attempts to violate a memory protection set by a higher VTL, an intercept is generated. This intercept is received by the higher VTL which set the protection. This allows higher VTLs to deal with the violation on a case-by-case basis. For example, the higher VTL may choose to return a fault, or emulate the access.

Mode Based Execute Control (MBEC)

When a VTL places a memory restriction on a lower VTL, it may wish to make a distinction between user and kernel mode when granting an “execute” privilege. For example, if code integrity checks were to take place in a higher VTL, the ability to distinguish between user-mode and kernel-mode would mean that a VTL could enforce code integrity for only kernel-mode applications.

Apart from the traditional three memory protections (read, write, execute), MBEC introduces a distinction between user-mode and kernel-mode for execute protections. Thus, if MBEC is enabled, a VTL has the opportunity to set four types of memory protections:

Name	Description

Name	Description
Read	Controls whether read access is allowed to a memory page
Write	Controls whether write access allowed to a memory page
User Mode Execute (UMX)	Controls whether instruction fetches generated in user-mode are allowed for a memory page. NOTE: If MBEC is disabled, this setting is ignored.
Kernel Mode Execute (UMX)	Controls whether instruction fetches generated in kernel-mode are allowed for a memory page. NOTE: If MBEC is disabled, this setting controls both user-mode and kernel-mode execute accesses.

Memory marked with the “User-Mode Execute” protections would only be executable when the virtual processor is running in user-mode. Likewise, “Kernel-Mode Execute” memory would only be executable when the virtual processor is running in kernel-mode.

KMX and UMX can be independently set such that execute permissions are enforced differently between user and kernel mode. All combinations of UMX and KMX are supported, except for KMX=1, UMX=0. The behavior of this combination is undefined.

MBEC is disabled by default for all VTLs and virtual processors. When MBEC is disabled, the kernel-mode execute bit determines memory access restriction. Thus, if MBEC is disabled, KMX=1 code is executable in both kernel and user-mode.

Descriptor Tables

Any user-mode code that accesses descriptor tables must be in GPA pages marked as KMX=UMX=1. User-mode software accessing descriptor tables from a GPA page marked KMX=0 is unsupported and results in a general protection fault.

MBEC configuration

To make use of Mode-based execution control, it must be enabled at two levels:

1. When the VTL is enabled for a partition, MBEC must be enabled using HvCallEnablePartitionVtl
2. MBEC must be configured on a per-VP and per-VTL basis, using HvRegisterVsmVpSecureVtlConfig.

MBEC Interaction with Supervisor Mode Execution Prevention (SMEP)

Supervisor-Mode Execution Prevention (SMEP) is a processor feature supported on some platforms. SMEP can impact the operation of MBEC due to its restriction of supervisor access to memory pages. The hypervisor adheres to the following policies related to SMEP:

- If SMEP is not available to the guest OS (whether it be because of hardware capabilities or processor compatibility mode), MBEC operates unaffected.
- If SMEP is available, and is enabled, MBEC operates unaffected.
- If SMEP is available, and is disabled, all execute restrictions are governed by the KMX control. Thus, only code marked KMX=1 will be allowed to execute.

Virtual Processor State Isolation

Virtual processors maintain separate states for each active VTL. However, some of this state is private to a particular VTL, and the remaining state is shared among all VTLs.

State which is preserved per VTL (a.k.a. private state) is saved by the hypervisor across VTL transitions. If a VTL switch is initiated, the hypervisor saves the current private state for the active VTL, and then switches to the private state of the target VTL. Shared state remains active regardless of VTL switches.

Private State

In general, each VTL has its own control registers, RIP register, RSP register, and MSRs. Below is a list of specific registers and MSRs which are private to each VTL.

Private MSRs:

- SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP, STAR, LSTAR, CSTAR, SFMASK, EFER, PAT, KERNEL_GSBASE, FS.BASE, GS.BASE, TSC_AUX
- HV_X64_MSR_HYPERCALL
- HV_X64_MSR_GUEST_OS_ID
- HV_X64_MSR_REFERENCE_TSC
- HV_X64_MSR_APIC_FREQUENCY
- HV_X64_MSR_EOI
- HV_X64_MSR_ICR
- HV_X64_MSR_TPR
- HV_X64_MSR_APIC_ASSIST_PAGE
- HV_X64_MSR_NPIEP_CONFIG

- HV_X64_MSR_SIRBP
- HV_X64_MSR_SCONTROL
- HV_X64_MSR_SVERSION
- HV_X64_MSR_SIEFP
- HV_X64_MSR_SIMP
- HV_X64_MSR_EOM
- HV_X64_MSR_SINT0 – HV_X64_MSR_SINT15
- HV_X64_MSR_STIMER0_CONFIG – HV_X64_MSR_STIMER3_CONFIG
- HV_X64_MSR_STIMER0_COUNT – HV_X64_MSR_STIMER3_COUNT
- Local APIC registers (including CR8/TPR)

Private registers:

- RIP, RSP
- RFLAGS
- CR0, CR3, CR4
- DR7
- IDTR, GDTR
- CS, DS, ES, FS, GS, SS, TR, LDTR
- TSC
- DR6 (*dependent on processor type. Read HvRegisterVsmCapabilities virtual register to determine shared/private status)

Shared State

VTLs share state in order to cut down on the overhead of switching contexts. Sharing state also allows some necessary communication between VTLs. Most general purpose and floating point registers are shared, as are most architectural MSRs. Below is the list of specific MSRs and registers that are shared among all VTLs:

Shared MSRs:

- HV_X64_MSR_TSC_FREQUENCY
- HV_X64_MSR_VP_INDEX
- HV_X64_MSR_VP_RUNTIME
- HV_X64_MSR_RESET
- HV_X64_MSR_TIME_REF_COUNT
- HV_X64_MSR_GUEST_IDLE
- HV_X64_MSR_DEBUG_DEVICE_OPTIONS
- MTRRs
- MCG_CAP
- MCG_STATUS

Shared registers:

- Rax, Rbx, Rcx, Rdx, Rsi, Rdi, Rbp
- CR2
- R8 – R15
- DR0 – DR3
- X87 floating point state
- XMM state
- AVX state
- XCR0 (XFEM)
- DR6 (*dependent on processor type. Read HvRegisterVsmCapabilities virtual register to determine shared/private status)

Real Mode

Real mode is not supported for any VTL greater than 0. VTLs greater than 0 can run in 32-bit or 64-bit mode.

VTL Interrupt Management

In order to achieve a high level of isolation between Virtual Trust Levels, Virtual Secure Mode provides a separate interrupt subsystem for each VTL enabled on a virtual processor. This ensures that a VTL is able to both send and receive interrupts without interference from a less secure VTL.

Each VTL has its own interrupt controller, which is only active if the virtual processor is running in that particular VTL. If a virtual processor switches VTL states, the interrupt controller active on the processor is also switched.

An interrupt targeted at a VTL which is higher than the active VTL will cause an immediate VTL switch. The higher VTL can then receive the interrupt. If the higher VTL is unable to receive the interrupt because of its TPR/CR8 value, the interrupt is held as "pending" and the VTL does not switch. If there are multiple VTLs with pending interrupts, the highest VTL takes precedence (without notice to the lower VTL).

When an interrupt is targeted at a lower VTL, the interrupt is not delivered until the next time the virtual processor transitions into the targeted VTL. INIT and startup IPIs targeted at a lower VTL are dropped on a virtual processor with a higher VTL enabled. Since INIT/SIPI is blocked, the HvCallStartVirtualProcessor hypercall should be used to start processors.

RFLAGS.IF

For the purposes of switching VTLs, RFLAGS.IF does not affect whether a secure interrupt triggers a VTL switch. If RFLAGS.IF is cleared to mask interrupts, interrupts into higher VTLs will still cause a VTL switch to a higher VTL. Only the higher VTL's TPR/CR8 value is taken into account when deciding whether to immediately interrupt.

This behavior also affects pending interrupts upon a VTL return. If the RFLAGS.IF bit is cleared to mask interrupts in a given VTL, and the VTL returns (to a lower VTL), the hypervisor will reevaluate any pending interrupts. This will cause an immediate call back to the higher VTL.

Virtual Interrupt Notification Assist

Higher VTLs may register to receive a notification if they are blocking immediate delivery of an interrupt to a lower VTL of the same virtual processor. Higher VTLs can enable Virtual Interrupt Notification Assist (VINA) via a virtual register HvRegisterVsmVina:

```c
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 Vector : 8;
        UINT64 Enabled : 1;
        UINT64 AutoReset : 1;
        UINT64 AutoEoi : 1;
        UINT64 ReservedP : 53;
    };
} HV_REGISTER_VSM_VINA;
```

Each VTL on each VP has its own VINA instance, as well as its own version of HvRegisterVsmVina. The VINA facility will generate an edge triggered interrupt to the currently active higher VTL when an interrupt for the lower VTL is ready for immediate delivery.

In order to prevent a flood of interrupts occurring when this facility is enabled, the VINA facility includes some limited state. When a VINA interrupt is generated, the VINA facility's state is changed to "Asserted." Sending an end-of-interrupt to the SINT associated with the VINA facility will not clear the "Asserted" state. The asserted state can only be cleared in one of two ways:

1. The state can manually be cleared by writing to the VinaAsserted field of the HV_VP_VTL_CONTROL structure.
2. The state is automatically cleared on the next entry to the VTL if the “auto-reset on VTL entry” option is enabled in the HvRegisterVsmVina register.

This allows code running at a secure VTL to just be notified of the first interrupt that is received for a lower VTL. If a secure VTL wishes to be notified of additional interrupts, it can clear the VinaAsserted field of the VP assist page, and it will be notified of the next new interrupt.

Secure Intercepts

The hypervisor allows a higher VTL to install intercepts for events that take place in the context of a lower VTL. This gives higher VTLs an elevated level of control over lower-VTL resources. Secure intercepts can be used to protect system-critical resources, and prevent attacks from lower-VTLs.

A secure intercept is queued to the higher VTL, and that VTL is made runnable on the VP.

Secure Intercept Types

Intercept Type	Intercept Applies To
Memory access	Attempting to access GPA protections established by a higher VTL.
Control register access	Attempting to access a set of control registers specified by a higher VTL.

Nested Intercepts

Multiple VTLs can install secure intercepts for the same event in a lower VTL. Thus, a hierarchy is established to decide where nested intercepts are notified. The following list is the order of where intercept are notified:

1. Lower VTL
2. Higher VTL

Handling Secure Intercepts

Once a VTL has been notified of a secure intercept, it must take action such that the lower VTL can continue. The higher VTL can handle the intercept in a number of ways, including: injecting an exception, emulating the access, or providing a proxy to the

access. In any case, if the private state of the lower VTL VP needs to be modified, HvCallSetVpRegisters should be used.

Secure Register Intercepts

A higher VTL can intercept on accesses to certain control registers. This is achieved by setting HvX64RegisterCrInterceptControl using the HvCallSetVpRegisters hypercall. Setting the control bit in HvX64RegisterCrInterceptControl will trigger an intercept for every access of the corresponding control register.

```c
typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 Cr0Write : 1;
        UINT64 Cr4Write : 1;
        UINT64 XCr0Write : 1;
        UINT64 IA32MiscEnableRead : 1;
        UINT64 IA32MiscEnableWrite : 1;
        UINT64 MsrLstarRead : 1;
        UINT64 MsrLstarWrite : 1;
        UINT64 MsrStarRead : 1;
        UINT64 MsrStarWrite : 1;
        UINT64 MsrCstarRead : 1;
        UINT64 MsrCstarWrite : 1;
        UINT64 ApicBaseMsrRead : 1;
        UINT64 ApicBaseMsrWrite : 1;
        UINT64 MsrEferRead : 1;
        UINT64 MsrEferWrite : 1;
        UINT64 GdtrWrite : 1;
        UINT64 IdtrWrite : 1;
        UINT64 LdtrWrite : 1;
        UINT64 TrWrite : 1;
        UINT64 MsrSysenterCsWrite : 1;
        UINT64 MsrSysenterEipWrite : 1;
        UINT64 MsrSysenterEspWrite : 1;
        UINT64 MsrSfmaskWrite : 1;
        UINT64 MsrTscAuxWrite : 1;
        UINT64 MsrSgxLaunchControlWrite : 1;
        UINT64 RsvdZ : 39;
    };
} HV_REGISTER_CR_INTERCEPT_CONTROL;
```

Mask Registers

To allow for finer control, a subset of control registers also have corresponding mask registers. Mask registers can be used to install intercepts on a subset of the corresponding control registers. Where a mask register is not defined, any access (as defined by HvX64RegisterCrInterceptControl) will trigger an intercept.

The hypervisor supports the following mask registers: HvX64RegisterCrInterceptCr0Mask, HvX64RegisterCrInterceptCr4Mask and HvX64RegisterCrInterceptIa32MiscEnableMask.

DMA and Devices

Devices effectively have the same privilege level as VTL0. When VSM is enabled, all device-allocated memory is marked as VTL0. Any DMA accesses have the same privileges as VTL0.

Nested Virtualization

Article • 03/17/2023

Nested virtualization refers to the Hyper-V hypervisor emulating hardware virtualization extensions. These emulated extensions can be used by other virtualization software (e.g. a nested hypervisor) to run on the Hyper-V platform.

This capability is only available to guest partitions. It must be enabled per virtual machine. Nested virtualization is not supported in a Windows root partition.

The following terminology is used to define various levels of nested virtualization:

Term	Definition
L0 Hypervisor	The Hyper-V hypervisor running on physical hardware.
L1 Root	The Windows root operating system.
L1 Guest	A Hyper-V virtual machine without a nested hypervisor.
L1 Hypervisor	A nested hypervisor running within a Hyper-V virtual machine.
L2 Root	A root Windows operating system, running within the context of a Hyper-V virtual machine.
L2 Guest	A nested virtual machine, running within the context of a Hyper-V virtual machine.

Compared to bare-metal, hypervisors can incur a significant performance regression when running in a VM. L1 hypervisors can be optimized to run in a Hyper-V VM by using enlightened interfaces provided by the L0 hypervisor.

Enlightened VMCS (Intel)

On Intel platforms, virtualization software uses virtual machine control data structures (VMCSs) to configure processor behavior related to virtualization. VMCSs must be made active using a VMPTRLD instruction and modified using VMREAD and VMWRITE instructions. These instructions are often a significant bottleneck for nested virtualization because they must be emulated.

The hypervisor exposes an “enlightened VMCS” feature which can be used to control virtualization-related processor behavior using a data structure in guest physical memory. This data structure can be modified using normal memory access instructions,

thus there is no need for the L1 hypervisor to execute VMREAD or VMWRITE or VMPTRLD instructions.

The L1 hypervisor may choose to use enlightened VMCSs by writing 1 to the corresponding field in the Virtual Processor Assist Page. Another field in the VP assist page controls the currently active enlightened VMCS. Each enlightened VMCS is exactly one page (4 KB) in size and must be initially zeroed. No VMPTRLD instruction must be executed to make an enlightened VMCS active or current.

After the L1 hypervisor performs a VM entry with an enlightened VMCS, the VMCS is considered active on the processor. An enlightened VMCS can only be active on a single processor at the same time. The L1 hypervisor can execute a VMCLEAR instruction to transition an enlightened VMCS from the active to the non-active state. Any VMREAD or VMWRITE instructions while an enlightened VMCS is active is unsupported and can result in unexpected behavior.

The HV_VMX_ENLIGHTENED_VMCS structure defines the layout of the enlightened VMCS. All non-synthetic fields map to an Intel physical VMCS encoding.

Clean Fields

The L0 hypervisor may choose to cache parts of the enlightened VMCS. The enlightened VMCS clean fields control which parts of the enlightened VMCS are reloaded from guest memory on a nested VM entry. The L1 hypervisor must clear the corresponding VMCS clean fields every time it modifies the enlightened VMCS, otherwise the L0 hypervisor might use a stale version.

The clean fields enlightenment is controlled via the synthetic “CleanFields” field of the enlightened VMCS. By default, all bits are set such that the L0 hypervisor must reload the corresponding VMCS fields for each nested VM entry.

Feature Discovery

Support for an enlightened VMCS interface is reported with CPUID leaf 0x40000004.

The enlightened VMCS structure is versioned to account for future changes. Each enlightened VMCS structure contains a version field, which is reported by the L0 hypervisor.

The only VMCS version currently supported is 1.

Hypervisor Implementation Considerations

For most VMCS fields, the corresponding enlightened VMCS field is supported for a VM if the VMCS field is supported for the VM, as determined through architectural feature discovery mechanisms. Exceptions are reported in CPUID leaf 0x4000000A.

In cases where architectural feature discovery mechanisms indicate support for a VMCS field for which no enlightened VMCS field is defined, the L1 hypervisor should not enable the feature if it chooses to use enlightened VMCS.

The Hyper-V L0 hypervisor will not indicate support for a VMCS field for which no enlightened VMCS field or exception is defined. If another L0 hypervisor needs a new enlightened VMCS field or exception to be defined, please contact Microsoft.

Enlighened VMCB fields (AMD)

AMD has reserved space in the VMCB for hypervisor use, as well as an associated clean bit. The reserved bytes are in the control section, offset 0x3E0-3FF, of the VMCB. The clean bit is bit 31 (the clean bit must be invalidated whenever the hypervisor modifies the “enlightenments” area of the VMCB).

Hyper-V utilizes the reserved VMCB area to configure enlightenments. The HV_SVM_ENLIGHTENED_VMCB_FIELDS structure documents the format.

Enlightened MSR Bitmap

The L0 hypervisor emulates the “MSR-Bitmap” controls on both Intel and AMD platforms that allow virtualization software to control which MSR accesses generate intercepts.

The L1 hypervisor may collaborate with the L0 hypervisor to make MSR accesses more efficient. It can enable enlightened MSR bitmaps by setting the corresponding field in the enlightened VMCS / VMCB fields to 1. When enabled, the L0 hypervisor does not monitor the MSR bitmaps for changes. Instead, the L1 hypervisor must invalidate the corresponding clean field after making changes to one of the MSR bitmaps.

Support for the enlightened MSR bitmap is reported in CPUID leaf 0x4000000A.

Compatibility with Live Migration

Hyper-V has the ability to live migrate a child partition from one host to another host. Live migrations are typically transparent to the child partition. However, in the case of nested virtualization, the L1 hypervisor may need to be aware of migrations.

Live Migration Notification

An L1 hypervisor can request to be notified when its partition is migrated. This capability is enumerated in CPUID as "AccessReenlightenmentControls" privilege. The L0 hypervisor exposes a synthetic MSR (HV_X64_MSR_REENLIGHTENMENT_CONTROL) that may be used by the L1 hypervisor to configure an interrupt vector and target processor. The L0 hypervisor will inject an interrupt with the specified vector after each migration.

```c
#define HV_X64_MSR_REENLIGHTENMENT_CONTROL (0x40000106)

typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 Vector :8;
        UINT64 RsvdZ1 :8;
        UINT64 Enabled :1;
        UINT64 RsvdZ2 :15;
        UINT64 TargetVp :32;
    };
} HV_REENLIGHTENMENT_CONTROL;
```

The specified vector must correspond to a fixed APIC interrupt. TargetVp specifies the virtual processor index.

TSC Emulation

A guest partition may be live migrated between two machines with different TSC frequencies. In those cases, the TscScale value from the reference TSC page may need to be recomputed.

The L0 hypervisor optionally emulates all TSC accesses after a migration until the L1 hypervisor has had the opportunity to recompute the TscScale value. The L1 hypervisor can opt into TSC Emulation by writing to the HV_X64_MSR_TSC_EMULATION_CONTROL MSR. If opted in, the L0 hypervisor emulates TSC accesses after a migration takes place.

The L1 hypervisor can query if TSC accesses are currently being emulated using the HV_X64_MSR_TSC_EMULATION_STATUS MSR. For example, the L1 hypervisor could subscribe to Live Migration notifications and query the TSC status after it receives the migration interrupt. It can also turn off TSC emulation (after it updates the TscScale value) using this MSR.

C

```c
#define HV_X64_MSR_TSC_EMULATION_CONTROL (0x40000107)
#define HV_X64_MSR_TSC_EMULATION_STATUS (0x40000108)

typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 Enabled :1;
        UINT64 RsvdZ :63;
    };
 } HV_TSC_EMULATION_CONTROL;

typedef union
{
    UINT64 AsUINT64;
    struct
    {
        UINT64 InProgress : 1;
        UINT64 RsvdP1 : 63;
    };
} HV_TSC_EMULATION_STATUS;
```

Virtual TLB

The virtual TLB exposed by the hypervisor may be extended to cache translations from L2 GPAs to GPAs. As with the TLB on a logical processor, the virtual TLB is a non-coherent cache, and this non-coherence is visible to guests. The hypervisor exposes operations to manage the TLB.

Direct Virtual Flush

The hypervisor exposes hypercalls (HvCallFlushVirtualAddressSpace, HvCallFlushVirtualAddressSpaceEx, HvCallFlushVirtualAddressList, and HvCallFlushVirtualAddressListEx) that allow operating systems to more efficiently manage the virtual TLB. The L1 hypervisor can choose to allow its guest to use those hypercalls and delegate the responsibility of handling them to the L0 hypervisor. This requires the use of a partition assist page (and a virtual VMCS on Intel platforms).

When in use, the virtual TLB tags all cached mappings with an identifier of the nested context (VMCS or VMCB) that created them. In response to a direct virtual flush hypercall from a L2 guest, the L0 hypervisor invalidates all cached mappings created by nested contexts where

- The VmId is the same as the caller's VmId
- Either the VpId is contained in the specified ProcessorMask or HV_FLUSH_ALL_PROCESSORS is specified

Support for direct virtual flushes is reported in CPUID leaf 0x4000000A.

Configuration

Direct handling of virtual flush hypercalls is enabled by:

1. Setting the NestedEnlightenmentsControl.Features.DirectHypercall field of the Virtual Processor Assist Page to 1.
2. Setting the EnlightenmentsControl.NestedFlushVirtualHypercall field of an enlightened VMCS or VMCB to 1.

Before enabling it, the L1 hypervisor must configure the following additional fields of the enlightened VMCS / VMCB:

- VpId: ID of the virtual processor that the enlightened VMCS / VMCB controls.
- VmId: ID of the virtual machine that the enlightened VMCS / VMCB belongs to.
- PartitionAssistPage: Guest physical address of the partition assist page.

The L1 hypervisor must also expose the following capabilities to its guests via CPUID.

- UseHypercallForLocalFlush
- UseHypercallForRemoteFlush

Partition Assist Page

The partition assist page is a page-size aligned page-size region of memory that the L1 hypervisor must allocate and zero before direct flush hypercalls can be used. Its GPA must be written to the corresponding field in the enlightened VMCS / VMCB.

C

```c
struct
{
    UINT32 TlbLockCount;
} VM_PARTITION_ASSIST_PAGE;
```

Synthetic VM-Exit

If the TlbLockCount of the caller's partition assist page is non-zero, the L0 hypervisor delivers a VM-Exit with a synthetic exit reason to the L1 hypervisor after handling a direct virtual flush hypercall.

On Intel platforms, a VM-Exit with exit reason `HV_VMX_SYNTHETIC_EXIT_REASON_TRAP_AFTER_FLUSH` is delivered. On AMD platforms, a VM-Exit with exit code `HV_SVM_EXITCODE_ENL` is delivered and ExitInfo1 is set to `HV_SVM_ENL_EXITCODE_TRAP_AFTER_FLUSH`.

```c
#define HV_VMX_SYNTHETIC_EXIT_REASON_TRAP_AFTER_FLUSH 0x10000031

#define HV_SVM_EXITCODE_ENL 0xF0000000
#define HV_SVM_ENL_EXITCODE_TRAP_AFTER_FLUSH   (1)
```

Second Level Address Translation

When nested virtualization is enabled for a guest partition, the memory management unit (MMU) exposed by the partition includes support for second level address translation. Second level address translation is a capability that can be used by the L1 hypervisor to virtualize physical memory. When in use, certain addresses that would be treated as guest physical addresses (GPAs) are treated as L2 guest physical addresses (L2 GPAs) and translated to GPAs by traversing a set of paging structures.

The L1 hypervisor can decide how and where to use second level address spaces. Each second level address space is identified by a guest defined 64-bit ID value. On Intel platforms, this value is the same as the EPT pointer. On AMD platforms, the value equals the nCR3 VMCB field.

Compatibility

The second level address translation capability exposed by the hypervisor is generally compatible with VMX or SVM support for address translation. However, the following guest-observable differences exist:

- Internally, the hypervisor may use shadow page tables that translate L2 GPAs to SPAs. In such implementations, these shadow page tables appear to software as large TLBs. However, several differences may be observable. First, shadow page tables can be shared between two virtual processors, whereas traditional TLBs are per-processor structures and are independent. This sharing may be visible because

a page access by one virtual processor can fill a shadow page table entry that is subsequently used by another virtual processor.

- Some hypervisor implementations may use internal write protection of guest page tables to lazily flush MMU mappings from internal data structures (for example, shadow page tables). This is architecturally invisible to the guest because writes to these tables will be handled transparently by the hypervisor. However, writes performed to the underlying GPA pages by other partitions or by devices may not trigger the appropriate TLB flush.
- On some hypervisor implementations, a second level page fault might not invalidate cached mappings.

Enlightened Second Level TLB Flushes

The hypervisor also supports a set of enhancements that enable a guest to manage the second level TLB more efficiently. These enhanced operations can be used interchangeably with legacy TLB management operations.

The hypervisor supports the following hypercalls to invalidate TLBs:

Hypercall	Description
HvCallFlushGuestPhysicalAddressSpace	invalidates cached L2 GPA to GPA mappings within a second level address space.
HvCallFlushGuestPhysicalAddressList	invalidates cached GVA / L2 GPA to GPA mappings within a portion of a second level address space.

On AMD platforms, all TLB entries are architecturally tagged with an ASID (address space identifier). Invalidation of the ASID causes all TLB entires associated with the ASID to be invalidated. The nested hypervisor can optionally opt into an "enlightened TLB" by setting EnlightenedNptTlb to "1" in HV_SVM_ENLIGHTENED_VMCB_FIELDS. If the nested hypervisor opts into the enlightenment, ASID invalidations just flush TLB entires derived from first level address translation (i.e. the virtual address space). To flush TLB entries derived from the nested page table (NPT) and force the L0 hypervisor to rebuild shadow page tables, the HvCallFlushGuestPhysicalAddressSpace or HvCallFlushGuestPhysicalAddressList hypercalls must be used.

Nested Virtualization Exceptions

On Intel platforms. the L1 hypervisor can opt in to combining virtualization exceptions in the page fault exception class. The L0 hypervisor advertises support for this enlightment in the Hypervisor Nested Virtualization Features CPUID leaf.

This enlightenment can be enabled by setting VirtualizationException to "1" in HV_NESTED_ENLIGHTENMENTS_CONTROL data structure in the Virtual Processor Assist Page.

Nested Virtualization MSRs

Nested VP Index Register

The L1 hypervisor exposes a MSR that reports the current processor's underlying VP index.

MSR address	Register Name	Description
0x40001002	HV_X64_MSR_NESTED_VP_INDEX	In a nested root partition, reports the current processor's underlying VP index.

Nested SynIC MSRs

In a nested root partition, the following MSRs allow access to the corresponding SynIC MSRs of the base hypervisor.

To find the index of the underlying processor, callers should first use HV_X64_MSR_NESTED_VP_INDEX.

MSR address	Register Name	Underlying MSR
0x40001080	HV_X64_MSR_NESTED_SCONTROL	HV_X64_MSR_SCONTROL
0x40001081	HV_X64_MSR_NESTED_SVERSION	HV_X64_MSR_SVERSION
0x40001082	HV_X64_MSR_NESTED_SIEFP	HV_X64_MSR_SIEFP
0x40001083	HV_X64_MSR_NESTED_SIMP	HV_X64_MSR_SIMP
0x40001084	HV_X64_MSR_NESTED_EOM	HV_X64_MSR_EOM
0x40001090	HV_X64_MSR_NESTED_SINT0	HV_X64_MSR_SINT0
0x40001091	HV_X64_MSR_NESTED_SINT1	HV_X64_MSR_SINT1
0x40001092	HV_X64_MSR_NESTED_SINT2	HV_X64_MSR_SINT2
0x40001093	HV_X64_MSR_NESTED_SINT3	HV_X64_MSR_SINT3
0x40001094	HV_X64_MSR_NESTED_SINT4	HV_X64_MSR_SINT4

MSR address	Register Name	Underlying MSR
0x40001095	HV_X64_MSR_NESTED_SINT5	HV_X64_MSR_SINT5
0x40001096	HV_X64_MSR_NESTED_SINT6	HV_X64_MSR_SINT6
0x40001097	HV_X64_MSR_NESTED_SINT7	HV_X64_MSR_SINT7
0x40001098	HV_X64_MSR_NESTED_SINT8	HV_X64_MSR_SINT8
0x40001099	HV_X64_MSR_NESTED_SINT9	HV_X64_MSR_SINT9
0x4000109A	HV_X64_MSR_NESTED_SINT10	HV_X64_MSR_SINT10
0x4000109B	HV_X64_MSR_NESTED_SINT11	HV_X64_MSR_SINT11
0x4000109C	HV_X64_MSR_NESTED_SINT12	HV_X64_MSR_SINT12
0x4000109D	HV_X64_MSR_NESTED_SINT13	HV_X64_MSR_SINT13
0x4000109E	HV_X64_MSR_NESTED_SINT14	HV_X64_MSR_SINT14
0x4000109F	HV_X64_MSR_NESTED_SINT15	HV_X64_MSR_SINT15

Hyper-V Backup Approaches

Article • 10/17/2022

Hyper-V allows you to backup virtual machines, from the host operating system, without the need to run custom backup software inside the virtual machine. There are several approaches that are available for developers to utilize depending on their needs.

Hyper-V VSS Writer

Hyper-V implements a VSS writer on all versions of Windows Server where Hyper-V is supported. This VSS writer allows developers to utilize the existing VSS infrastructure to backup virtual machines. However, it is designed for small scale backup operations where all virtual machines on a server are backed up simultaneously.

Hyper-V WMI Based Backup

Starting in Windows Server 2016, Hyper-V started supporting backup through the Hyper-V WMI API. This approach still utilizes VSS inside the virtual machine for backup purposes, but no longer uses VSS in the host operating system. Instead, a combination of reference points and resilient change tracking (RCT) is used to allow developers to access the information about backed up virtual machines in an efficient manner. This approach is more scalable than using VSS in the host, however it is only available on Windows Server 2016 and later.

There is also an example on how to use these APIs available here: https://www.powershellgallery.com/packages/xHyper-VBackup

Methods for reading backups from WMI Based Backup

When creating virtual machine backups using Hyper-V WMI, there are three methods for reading the actual data from the backup. Each has unique advantages and disadvantages.

WMI Export

Developers can export the backup data through the Hyper-V WMI interfaces (as used in the above example). Hyper-V will compile the changes into a virtual hard drive and copy

the file to the requested location. This method is easy to use, works for all scenarios and is remotable. However, the virtual hard drive generated often creates a large amount of data to transfer over the network.

Win32 APIs

Developers can use the SetVirtualDiskInformation, GetVirtualDiskInformation and QueryChangesVirtualDisk APIs on the Virtual Hard Disk Win32 API set as documented here. Note that to use these APIs, Hyper-V WMI still needs to be used to create reference points on associated virtual machines. These Win32 APIs then allow for efficient access to the data of the backed up virtual machine. The Win32 APIs do have several limitations:

- They can only be accessed locally
- They do not support reading data from shared virtual hard disk files
- They return data addresses that are relative to the internal structure of the virtual hard disk

Remote Shared Virtual Disk Protocol

Finally, if a developer needs to efficiently access backup data information from a shared virtual hard disk file – they will need to use the Remote Shared Virtual Disk Protocol. This protocol is documented here.